RIM: RECONFIGURABLE INSTRUCTION MEMORY HIERARCHY FOR EMBEDDED SYSTEMS

ZHIGUO, GE

NATIONAL UNIVERSITY OF SINGAPORE

2008

RIM: RECONFIGURABLE INSTRUCTION MEMORY HIERARCHY FOR EMBEDDED SYSTEMS

ZHIGUO, GE
(M.Eng., Zhejiang University)

A thesis submitted for the degree of Doctor of Philosophy
in Computer Science

Department of Computer Science

National University of Singapore

2008

Acknowledgement

I would like to thank my advisor, Professor Wong Weng-Fai, for his guidance and encouragement throughout my Phd study. He offered me this precious opportunity of Phd study and led me into this research area which interests me enormously. He provided me with freedom to conduct independent research while actively giving constructive comments, suggestions and important support. I benefited a lot from his knowledge, insight into research and as a generous and kind advisor, he always constantly helps students not only in academic development, but also in their lives.

I am very grateful to Professor Tulika Mitra for extensive discussions and for her invaluable advice. I am so impressed on her sincere attitude, intellect, and insight into research. This experience will continue to motivate me to do research and work sincerely in the rest of my life. I thank Professor Samarjit Chakraborty for many useful discussions and comments in refining the ideas in this dissertation.

I am indebt to my master degree advisor, Professor Zheng Jialong in Zhejiang University for his continuous trust and support. As a generous, wise and kind advisor, he always encourages and helps students like a father to realize their potentials. His support and encouragement are of crucial importance to me.

My sincere thanks are due to Dr. Lim Hock Beng for the collaboration and his helpful advice at the beginning of my Phd research.

I would like to thank the National University of Singapore for the research scholarship that makes this study possible and the administrative and technical staff here for their various support.

I would like to appreciate all the members in embedded systems lab for their kind help and the enjoyable life together. Special acknowledgements go to Yu Pan, Qin Zhao, Yanhong Liu, Kathy Nguyen Dang, Zhenxin Sun, Edward Sim, Yun Liang, Lei Ju, Ankit goel, Ramkumar Jayaseelan, Unmesh Dutta Bordoloi, Xiuchao Wu and Mingze Zhang for their support in both academic and my daily life.

Finally, I deeply thank my family for their constant support, encouragement, trust and love. My parents do whatever to avoid distracting me from study. My special thanks are due to my sisters and their family for their loving support. They help take care of my parents and thus I can study without concerns. Without their unconditional support, I cannot fulfill the Phd study.

Contents

Abstract

Embedded systems have been becoming increasingly popular during the past decades. Such systems are constrained by hardware resources and energy consumption but they are still required to maintain high performance. Instruction memory hierarchy is the bottleneck of both performance and energy consumption for embedded systems and researchers have been putting much effort to improve performance and reduce energy consumption for instruction memory hierarchy. Among them, customizing parameters of the instruction memory hierarchy for given applications is one of the most important methods.

In this thesis, we propose and design reconfigurable instruction memory hierarchies for embedded systems to achieve performance and energy improvement. Our work includes the following contributions:

1. We propose a static reconfigurable instruction memory hierarchy(SRIM). Instead of using pure cache, the given resource budget for instruction memory hierarchy is divided into certain size of scratchpad memory(SPM) and cache for a given application to maximize the performance and minimize energy consumption. We study the interaction between the architecture exploration of SRIM and instruction layout optimization for more energy savings.

2. Although SRIM provides more flexibilities to make better use of the given resource budget, it has two major limitations: 1.) it cannot reconfigured for

different phases in an execution 2.) the SPM resource cannot be reused by multiple instructions and this cause low efficiency of resource usage. To remedy the drawbacks of the static reconfigurable instruction memory hierarchy, we propose a dynamic reconfigurable instruction memory to obtain more flexibility. The static reconfigurable instruction memory hierarchy cannot change the hardware parameters for different execution phases during runtime.

3. We propose a DVS-based pipelined reconfigurable instruction memory hierarchy(PRIM) for low power embedded systems. One of the most common methods to decrease energy consumption is to shut down the under-utilized storage resources. Instead of shutting down under-utilized resources, our method tries to take advantage of under-utilized storage resources to achieve more energy savings.

Our proposed schemes can tune hardware parameters for specific applications and they are more flexible. As a result of this flexibility, they achieve great performance and energy improvement. In addition, we have implemented SRIM into real hardware using FPGA, which proves it is implementable into real systems.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Embedded systems

Embedded systems have been increasingly popular and widely used over the past decades. In contrast to general purpose computing systems, an embedded system is a special-purpose computer system that performs one or a few dedicated functions. Such systems range from small devices such as mobile phones, digital cameras, etc to large scale systems like factory controllers. They are widely used in our daily lives and the number of such systems has already surpassed the human population in this planet [96]. In the near future, the world will be more inter-connected by embedded systems such as sensor networks as new technologies become mature. In contrast to general purpose computing systems, they have several distinct characteristics listed below:

1. Most embedded systems run only one or a few fixed and dedicated applications. For example, a mp3 player's only functionality is to play music files in its entire life-time.

2. Unlike general computing systems, embedded systems are characterized by strict constrains on resources and low energy budget. This is especially true for portable devices where battery life is of crucial importance.

3. Despite these constraints, embedded systems are still expected to provide high computation capability and meet the real-time constraints.

These characteristics make embedded system very hard to design, as designers not only need to consider performance but also energy consumption and hardware resource usage. To tackle this problem, application-specific design methodology has been widely used in embedded systems where hardware and software optimizations are performed based on a given application. The parts of hardware design and software optimization are usually highly coupled and the hardware architecture is determined according to the feature of the given application. As the result of this design methodology, one prominent characteristic of embedded systems is the heterogeneous architecture with programmable processor cores, custom designed logic, and different types of memory. The architecture can be tailored or reconfigured for specific applications. Custom instructions intended for specific applications can dramatically improve the CPU performance. Tuning the cache size, associativity and line size can best make use of the limited hardware resources to achieve maximal performance for the application. For most cases, customizing the hardware architecture requires sophisticated compiler support for generating the corresponding code for the customized architecture. Thus, the compiler and hardware need to be managed together in the design of embedded systems.

The nature of the heterogeneous architecture and the tightly-coupled hardware and software of embedded systems gives rise many research issues involving architecture and software co-optimization. Among them, memory subsystem is one of the most severe concerns we need to deal with. Memory wall is a well-known problem

in computer architecture. The speed of processors has become increasingly faster, while the increase of the memory speed has not been able to keep pace with that of processors. And memory access time has become a bottleneck in computer systems. Moreover, memory hierarchy occupies large die area and consumes a lot of energy, both of which are very scarce for embedded systems. In this thesis, we study the instruction memory hierarchy designs for embedded systems.

1.2 Memory hierarchy issues

1.2.1 Memory wall problem

Over the past decades, microprocessor speed has been growing at a dramatic rate of 50-100% per year, while the speed of DRAM [66] is about 7% per year at the same period. As a consequence, the memory speed has become a bottleneck of whole system performance which is known as *memory wall problem* [36]. Apart from the performance bottleneck, the memory subsystem also occupies very large area, and thus consumes significant portion of total energy and demonstrates energy bottleneck because of the load capacitance. For example, the Itanium2, IBM G5, Power PC and Strong-ARM have 80%, 72%, 71% and 70% of total transistors in caches respectively [76]. The caches account for a large portion of the total power dissipation(e.g.,35% in Itanium; 43% in Strong-ARM;) [76]. The area and power of caches in embedded processor ARM1156T2-S take more than 41.4% and 22.6% of the total area and power [9].

Due to the large power consumption and the high access rate to memory subsystem, memory subsystem accounts for significant of overall energy consumption. Several researchers [49, 96] showed that the memory subsystem accounts for 50-70%

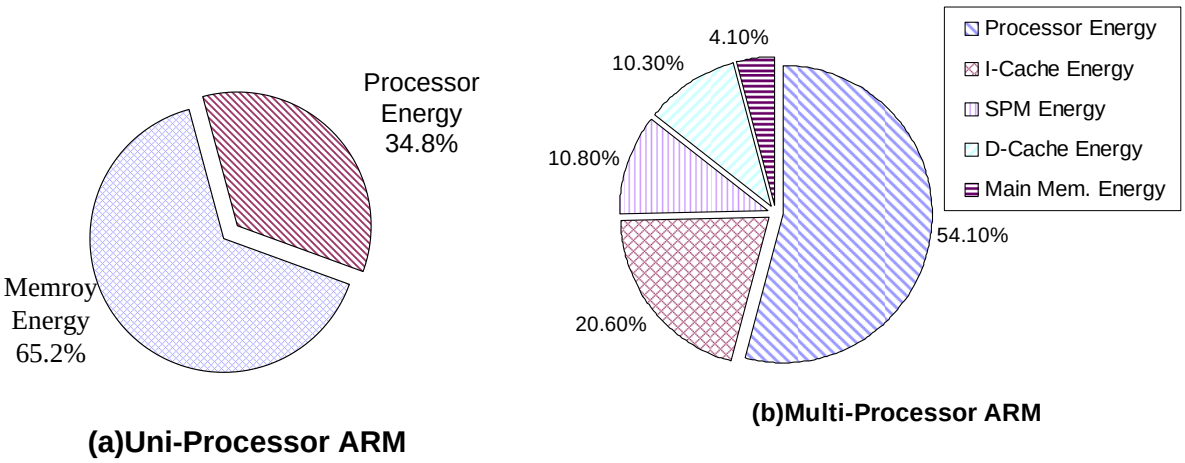

Figure 1.1: Energy distribution for two ARM based setups [96]

of the total energy consumption of a whole system. Figure 1.1 redrawn from [96] shows the energy consumption by different components for uni-processor ARM and multi-processor ARM based systems. The memory subsystem account for 65.2% and 45.9% of total energy consumption for uni-processor ARM and multi-processor ARM systems respectively. As seen from figure 1.1.(b), instruction cache accounts for a significant portion of the total energy(i.e., 20.6%).

Because of the crucial role memory subsystem plays on performance and energy, designers have to take great care of the memory subsystem to improve the performance and decrease energy consumption, especially for portable embedded systems. It is of crucial importance to improve the performance and energy consumption of instruction memory hierarchy for embedded systems.

1.3 Memory hierarchy design

Memory subsystem has received great attention during the past decades. There are basically two different categories of approaches to improve the performance and energy of the memory hierarchy. One is the software management and optimization; the other is the hardware design.

Many methods of application optimization have been developed to improve performance and reduce energy consumption. To improve performance, several loop transformations [101, 21] have been developed to improve data reuse, data locality, or decrease memory accesses, etc. In addition, instruction layout [75, 35, 29] and data layout [21, 71, 50] optimizations have been proposed to improve the memory locality to decrease cache conflict misses.

Apart from the software approaches, various hardware approaches have also been thoroughly studied. In contrast to the software methods, the hardware methods design specific hardware architecture for embedded applications and tune the hardware configuration for a given application. The hardware method is popular and applicable in embedded systems because usually there is only one or a few applications running on embedded systems and this offers many opportunities to custom hardware parameters for the given applications. A lot of cache architectures to reduce cache activities have been proposed for energy savings. For example, **Phased-lookup cache** [34], accesses cache tags first and then the data bank corresponding to the hit tag, and as a result only one data bank needs to be searched. **Pseudo set-associative caches (PSAC)** [52, 79, 43, 20, 42] speculatively accesses one cache bank in the hope to hit to save energy. Small buffers [58, 62, 30] with extremely low energy consumption are added between L1 cache and CPU core to decrease energy consumption. Studies [86, 87] show that the execution characteristics vary across different applications and even phases within an execution. Thus, the hardware requirement to cache changes dramatically. To meet this need, reconfigurable cache architectures [103, 95, 44, 80, 53, 28, 60, 13] have been proposed to improve performance and decrease energy consumption. The associativity [103] and line size of a cache [95] can be configured and furthermore cache resources can be configured as other type function units such as instruction buffers or computational units [80, 53, 28, 60, 13]. These methods try to disable the under-utilized resources

to save energy. In this thesis, we introduce a new memory hierarchy, namely DVS-based pipelined reconfigurable instruction memory hierarchy, to take advantage of the under-utilized resources to obtain more energy savings.

Apart from variants of cache architectures, scratchpad memory (SPM) is a very important alternative of caches widely used in embedded systems. The difference between them is that the SPM does not have the extra tags and the control logic circuits. The applications control the SPM directly. Because of the absence of the tags and control logic circuits, SPM consumes much less energy than cache. Angiolini et al. [8] proposed an algorithm which statically partitions the instructions to SPM to save energy. The algorithm takes the profiling information as inputs and applies dynamic programming method to get the optimal instruction partitioning to SPM with regard to the energy savings. Because the method is static, the storage locations of SPM cannot be reused by other instructions once the partitioning has been determined. Consequently, the efficiency of SPM usage could be very low. More recently, a dynamic method [93] was proposed to remedy the restrictions of the static one. The instructions in the SPM can be dynamically replaced. The dynamic method offers the application the ability of replacing the instructions residing in SPM by other more frequently executed instructions during runtime. They reported 20%-30% improvement of energy savings over the static method. The above two methods assume the SPM parameters are fixed. However, different applications have their own characteristics, which in turn require different hardware configurations to maximize performance and minimize energy consumption. To the best of our knowledge, there is little work on configuring parameters of SPM for instruction memory for a given application. This thesis focuses on customizing the hardware parameters of SPM for instruction memory hierarchy for given applications to improve performance and reduce energy consumption.

1.4 Motivation and contributions

The main aim of this thesis is to investigate the static and dynamic reconfigurable instruction memory to improve performance and reduce energy consumption of the instruction hierarchy for embedded systems. It is known that execution characteristics vary across different applications and different phases of the application, which tend to result in different hardware requirements [86, 87]. Reconfiguring the memory hierarchy had been proven effective to improve performance and save energy[80, 103] by changing the memory hardware parameters to exploit this variation. Our research focuses on customizing the instruction memory hierarchy for given applications to reduce energy consumption and improve performance. Existing methods using SPM do not change the parameters of SPM to exploit this variation. In contrast to previous methods, we attempt to tune the hardware parameters of SPM according to needs of different applications and different phases of the given application for maximizing performance and reducing energy consumption. We investigate the following three problems which can be classified into static and dynamic reconfigurable instruction memory hierarchy designs.

1. SRIM: Static Reconfigurable Instruction Memory. To exploit the variation between different applications, we propose a static reconfigurable instruction memory hierarchy for embedded applications. Given an application and fixed resource budget for instruction memory hierarchy, we try to statically determine the best configuration of instruction memory hierarchy in terms of the size of SPM and cache for the given application. We profile the application and gather the statistics, such as basic block execution counts and function call counts. Analyzing the statistics, the best configuration in terms of cache and SPM size for the application is determined and meanwhile the instructions are partitioned to SPM and cache respectively. By applying this method, we at-

tempt to use the given resource more efficiently, and thus achieve performance improvement and energy savings.

2. DRIM: Dynamic Reconfigurable Instruction Memory. Although SRIM is able to explore the variation across applications, it cannot handle the variation of phases within an execution as the hardware parameters are fixed for an application. To tackle this problem, we propose a dynamic reconfigurable instruction memory hierarchy (i.e., DRIM) to explore the phaseal variation for energy reduction. The hardware parameters of DRIM can be dynamically configured according to the needs of phases at runtime. DRIM consists of four banks of on-chip instruction buffer, each of which can be dynamically configured as cache or SPM. The configuration of DRIM in terms of banks of SPM and caches thus can be changed at runtime. We profile the application, analyze the profiles, and determine the suitable configuration for each phase of the execution.

3. PRIM: A DVS-based Pipelined Reconfigurable Instruction Memory. Shutting down idle cache banks is one of the most popular methods to reduce the energy consumption. However, we observe that operating an idle cache bank I at reduced voltage/frequency level along with an active bank A can potentially achieve better energy savings compared to shutting down I and operating A in normal mode. The key to maintain the performance is to pipeline and synchronize the accesses to these two banks through the appropriate instruction memory layout. Moreover, static analysis should determine appropriate program phases where one can switch to low power mode to achieve significant energy savings with minimal performance overhead. Towards this, we propose a novel DVS-based pipelined reconfigurable instruction memory hierarchy called PRIM. A canonical example of our proposed instruction memory hierarchy consists of four cache banks. Two of these cache banks can be con-

figured at runtime to operate at half the voltage and frequency level of the normal cache mode. Instruction fetch throughput is maintained by pipelining the accesses to these two low voltage banks. We develop a profile-driven compilation framework that can analyze the application and insert appropriate cache reconfiguration points.

The main novelty of the work is that unlike previous works that disable the accessibility of idle cache banks, the capacity of idle banks in our scheme is exploited to help save more energy.

Our proposed architectures advance the research of this area because of their novelties in methodology. We have implemented the static instruction memory architecture (i.e.,SRIM) into real hardware and this has proven that the proposed architectural designs are practical for real electronics devices. Based on the static architecture introduced, our proposed dynamic architectures will be presented in the subsequent chapters. The compilation flow used to support the architectures has also been developed.

1.5 Organization of the thesis

The rest of the thesis is organized as follows:

Chapter 2: Background and related work. We introduce the background in memory hierarchy, and discuss previous and related work in memory hierarchy domain.

Chapter 3 :Static Reconfigurable Instruction Memory Hierarchy(SRIM). We describe the architecture of SRIM and the rational of SRIM to optimize the design for performance improvement and energy consumption reduction. We then present

the design and compilation flow to partition instructions to SPM and cache for maximizing performance and minimizing energy consumption.

Chapter 4: Integrated Memory Design. We integrate instruction layout optimization into SRIM for further performance improvement and energy reduction.

Chapter 5: Dynamic Reconfigurable Instruction Memory Hierarchy(DRIM). We present a dynamic reconfigurable instruction memory hierarchy and algorithm to dynamically reallocate resources to save more energy.

Chapter 6: This chapter presents a DVS-based Pipelined Reconfigurable Instruction Memory Hierarchy(PRIM) for low power embedded systems.

Chapter 7: Conclusions: We summarize the work presented in the thesis and point out possible directions for future work.

Chapter 2

Background and related work

Computer designers have been paying much attention to the memory subsystem because of its crucial role played in processor-based computation. First of all, the speed gap between main memory and processor has been increasing greatly. Second, each access to main memory is very energy expensive. Finally, each instruction execution involves one or more accesses to memory. The memory hierarchy is usually organized as several levels of memory where larger memory implies longer access latency.

There are basically two classes of methods, architectural designs and application optimizations, to improve the memory hierarchy performance and decrease energy consumption. From the architectural perspective, various cache architectures such as reconfigurable caches, victim cache, and pseudo set-associative caches are designed for performance improvement and energy reduction. On the other hand, from the software angle, designers can optimize the application to achieve the improvement based on fixed architecture. For example, loop transformations and instruction/data layout optimizations are developed to improve the data and instruction locality.

For embedded systems, jointly designing the architecture and optimizing the

software application is popular. This method can make use of limited resources more efficiently to achieve performance improvement and energy savings. Basically, memory hierarchy can be divided into two categories. One is data memory hierarchy, and the other is instruction memory hierarchy. Instructions are fetched at every cycle during the execution of applications. As a result, the instruction memory hierarchy can affect the performance and energy consumption dramatically. Our research focuses on reconfigurable instruction memory design and the goal is to improve the performance of instruction memory hierarchy and reduce the energy consumption.

With the proliferation of portable embedded devices, power consumption has become a major design consideration as power consumption affects the battery life and the heat dissipation of portable devices, which in turns affects their usability. Thus, designing an energy efficient memory subsystem is of crucial importance for such systems. To design low power memory subsystem, it is important to understand the source of power dissipation of memory systems.

2.1 Factors affecting power consumption of caches

In CMOS circuits of caches, there are two major components of power dissipation: dynamic power which is caused by the charge and discharge of capacitance, and static power which is due to leakage current I_{leak}. The dynamic power is proportional to the square of supply voltage, while static power is constant. The formula of total power of a storage block is as follow [57]:

$$P = ACV^2 f + V I_{leak} \tag{2.1}$$

The first term is the dynamic power. A is the fraction of actively switching gates and C is the total capacitance of all gates. V and f represent the supply voltage

and clock frequency respectively. The second term in Equation 2.1 is the static power and I_{leak} stands for the leakage current. As we can see from Equation 2.1, the dynamic energy consumption, ACV^2f, is proportional to circuit switching activities denoted by A. The more the memory are accessed, the more power it will consume. The static power denoted as VI_{leak} is independent on circuit switching activities.

The leakage current I_{leak}, source of static power, has two components: sub-threshold(I_{sub}) and gate-oxide leakage (I_{ox}). I_{sub} can be calculated by the following equation:

$$I_{sub} = K_1 W e^{-V_{th}/nV_\theta}(1 - e^{-V/V_\theta}) \tag{2.2}$$

Where the values of K_1 and n are experimentally derived, W is the width of gate, V_{th} and V_θ are the gate threshold voltage and thermal voltage respectively. Equation 2.2 suggests two methods to decrease I_{sub}: decrease or even shut down supply voltage and increase threshold voltage V_{th}.

The other component of leakage current can be calculated as follow [57]:

$$I_{ox} = K_2 W (\frac{V}{T_{ox}})^2 e^{-aT_{ox}/V} \tag{2.3}$$

Through the above formulas, we can conclude that decreasing supply voltage will reduce both dynamic and static power. However, as shown by the following equation, the speed of gate will be slowed down.

$$f \propto (V - V_{th})^\alpha / V = (1 - \frac{V_{th}}{V})^\alpha V^{\alpha-1} \tag{2.4}$$

The exponent α is an experimentally derived value which is approximately 1.3 for current technology. We can clearly see that clock frequency will be slowed when decreasing supply voltage V.

We can obtain the total energy consumption of memory hierarchy by summing up the energy consumption of all the levels of storages such as registers, L1 cache, L2 cache and main memory.

$$E_{total} = \sum_{all\ levels} Energy \tag{2.5}$$

2.2 Methods to decrease energy consumption

According to Equations 2.1, 2.2 and 2.5, we can derive the ways that can be used to decrease energy consumption of a memory hierarchy. We basically can classify these methods for reducing energy consumption into two categories: dynamic energy reduction and static energy reduction.

Dynamic energy reduction: Dynamic energy accounts for the most of the total energy consumption for current technology. Thus, most of the existing methods aim to reduce dynamic energy. From Equation 2.1, we can conclude that we can decrease energy consumption by the following measures:

- Reduce the fraction of switching circuit. According to the first component of Equation 2.1, dynamic power is proportional to the fraction of actively switching gates. Thus, preventing the unused or under-utilized resources from switching is one of the most important approaches to reduce dynamic power such as clock gating, pseudo-associative cache, etc.

- Reduce the capacitance of cache blocks. Apart from reducing the proportion of active gates, reducing the capacitance of cache blocks is another important approach for dynamic energy reduction. This scheme includes filter cache [58] whose size is extremely small and the resultant capacitance becomes small. As a result, the dynamic power is very low.

- Reduce supply voltage. As shown above, the dynamic energy is proportional to the square of V, the supply voltage. This implies that reducing V can greatly reduce dynamic power. Dynamic voltage scaling [31] takes advantage of this fact to decrease energy consumption. However, reducing V will also slow down the speed of circuits according to Equation 2.4. Consequently, this approach has to trade off the speed and power carefully.

- Reduce the access to next level of memory. The next level of memory is usually much larger which causes significant capacitance. Consequently the next level of memory storages is much more energy expensive. Because of this, reducing the number of accesses to lower level memories can also effectively decrease the total energy consumption of a memory hierarchy.

Static energy reduction: According to the static power formula, the second component of Equation 2.1, two factors affect the static power, voltage and leakage current. As a result, decreasing supply voltage and shutting under-utilized cache resources become very important methods to reduce static energy consumption, such as Gated-Vdd cache, cache decay, and cache drowsy cache.

2.3 Architectural methods

Much research work has been focusing on designing and optimizing the memory architecture for performance improvement and energy savings, especially for em-

bedded systems. The main idea is to tune memory architecture according to the needs of applications for the purpose of performance improvement and energy reduction. For example, special cache architectures, such as phased-lookup cache [34], pseudo set-associative [52] caches and filter cache [58], are designed to decrease energy consumption. Recently, researchers have been developing reconfigurable caches to dynamically adjust cache architectural parameters for performance improvement and energy savings. Apart from cache architectures, more energy efficient on-chip buffers such as scratchpad memory(SPM) [12] have been applied to save energy.

2.3.1 Cache architectures for reducing energy and improving performance

Several cache variations have been proposed to decrease energy consumption by reducing the activity of caches. **Phased-lookup cache** [34] is proposed to decrease the accesses to data banks in the cache for saving energy. The phased-lookup cache divides the cache access into two phases. First, all tags are searched using two-phase lookup where all the tag arrays are searched in parallel and none of data banks is accessed. In the next phase, the data bank corresponding to the tag hit is accessed. Through this way, only one data bank needs to be searched and thus energy can be saved. However, this sequential operation of cache access is only possible when address comparison determines a hit or miss before the data array starts its operation. This depends on the operating frequency.

As explained above, phased-lookup cache access to tag and data banks is sequential. And consequently this causes extra latency overhead of a cache access. **Pseudo set-associative caches (PSAC)** [52, 79, 43, 20, 42] are very important caches to obtain a good balance between energy and speed. The caches first predictively examine one tag and one data bank, and only access the other banks if the initial

access misses. Because the extra overhead of cache access latency is only incurred when prediction miss happens, the average access latency can be greatly reduced compared to the phased-lookup cache if the prediction hit rate is high. According to the way of probing the remaining banks after the first probe miss, PSAC can be classified into two categories: Fall Back Regula(FallBackReg) and Sequential [42]. **FallBackReg** scheme [79, 43] simultaneously probes all the remaining ways upon a initial probe miss while **Sequential scheme** [52, 20] sequentially probes all the remaining ways until all the data are found and an L1 miss is declared. FallBackReg scheme favors speed at the expense of energy, however, sequential scheme is more energy efficient at the expense of speed.

Another important method for decreasing cache activity is to add a tiny buffer in front of L1 cache, such as **filter cache** and **tiny cache**[58, 62, 30]. If the most of a program execution is spent in small loops, then most cache hit can occur in the tiny buffer. This can dramatically reduce accesses to L1 cache and hence effectively decrease energy consumption. The difference between filter cache and tiny cache is the cache control strategy: filter cache is hardware controlled while tiny cache is software controlled. Filter cache has a hardware component to dynamically detect loops at runtime and control the loop replacement. In contrast to filter cache, tiny cache needs compiler's support to analyze an application and statically decide the loop assignment.

Victim cache is proposed to reduce conflict cache misses for direct-mapped caches [48]. Direct-mapped cache is small, fast and low power. However, conflict misses account for significate portion, around 20% to 40% of total direct-mapped cache misses [48]. By adding a small full-associative victim cache and swapping the data between it and cache, cache conflict misses can be effectively reduced.

2.3.2 Variation between programs and phases of executions and reconfigurable cache

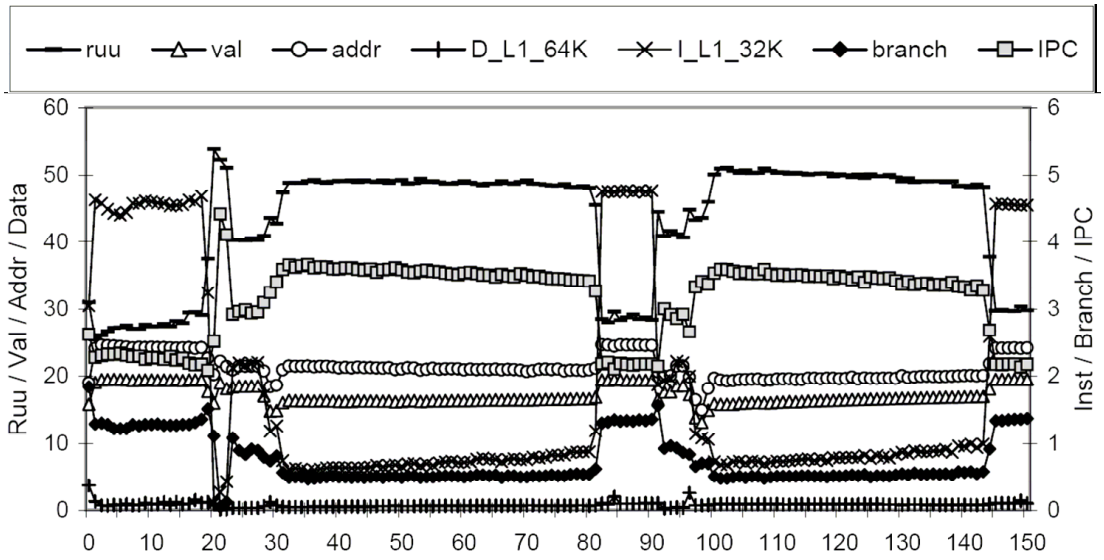

Figure 2.1: Time varying behavior for vortex

The rationale of the reconfigurable memory hierarchy is the variation of execution characteristics which include the variation between different programs and the phaseal variation within a program. A phase of program behavior can be defined as a time interval of execution during which a measured program metric is relatively stable [87]. Figure 2.1 excerpted from [86] shows the time behavior for vortex where *ruu* is percent Register Update Unit(RUU) occupancy while *val* and *addr* stand for value and address miss rates respectively. Instruction, data cache and branch prediction miss rates are denoted by D_L1_64K, I_L1_32K, *branch* respectively. IPC represents instructions per cycle. The X-axis is in terms of 100 million committed instructions. Left Y-axis is used by RUU, *val* and *addr* while D_L1_64K, I_L1_32K, *branch* and IPC use the right Y-axis. The figure clearly shows that the instruction miss rate varies dramatically at different time intervals as well as some other hardware metrics. Other than the variation within an execution, significant variation across different applications is also demonstrated [86]. Multiple methods to detect and predict phases of execution are proposed by Sherwood [87].

Reconfigurable caches are proposed to tune cache hardware parameters to exploit variations between phases and applications with the aim of improving performance and reducing energy consumption.

2.3.3 Customization of cache parameters

As described in the previous section, the execution characteristics vary across different applications and even within a single execution. As a result, a key issue related to customizing the cache parameters is the fact that the cache utilization varies widely for different applications and different time intervals. A number of studies have concentrated on dynamically adapting the cache geometry parameters, such as cache line and associativity, to save energy.

1. **Reducing cache size:** Unlike Pseudo set-associative caches which sequentially access cache banks, **selective way cache** [6] directly shuts down a subset of the ways in a set-associative cache for energy savings when the cache is under-utilized. The full cache remain operational for more cache-intensive periods. The decision of way shutting down is determined by the value Performance degradation Threshold(PDT) which signifies how much performance degradation relative to a cache with all ways enabled is allowable [6].

2. **Reconfiguration of cache associativity:** Using a new technique, called way concatenation [103], the cache associativity can be configured dynamically. The cache associativity greatly affects power consumption. Studies show that a direct-mapped cache consumes 30% energy of a same size four-way set associative cache [103]. The reason for the low power consumption of direct-mapped caches is that only one tag and one data array are read during an access, while four tags and four data arrays need to be read for four-way

associative cache. For some applications, a direct-mapped cache has a low miss rate, and thus results in a low energy consumption. However, some other applications may result in a high miss rate for direct-mapped cache. This high miss rate incurs larger energy consumption due to the longer execution time and more energy-consuming accesses to the larger low-level memory units. Thus, using a suitable cache associativity for a particular application is of great importance to decrease energy consumption.

3. **Reconfiguration of cache line size:** Similar to the cache associativity, the requirement for a suitable cache line size varies across different applications and different phases of an execution. Schemes to dynamically adjust cache line size are proposed to reduce the cache miss rate and improve the performance [95, 69, 44]. In [95], an automatic hardware system is used to monitor system behaviors and performance, and modify the configuration based on the observed behavior. In contrast to the hardware control method, in [69], compiler is used to insert configuration instructions where cache line size needs to be changed by either static code analysis or dynamic profiling-directed feedback.

4. **Reconfigurable cache architecture:** Other than configuring the parameters of caches, resources of cache itself can be reconfigured to other type of functional units, such as computational unit, during runtime [80, 53, 28, 60, 13]. A reconfigurable cache architecture for general-purpose processors is presented in [80]. The L1 data cache is partitioned into two banks. One of them is used as conventional cache, while the other can be configured as an instruction reuse buffer. For some multimedia applications, the data streaming characteristics cause a large cache under-utilized. In order to make better use of the hardware resources for such applications, one of the banks in the under-utilized cache can be configured into instruction reuse buffer. A multifunction computing cache

architecture is proposed in [53] where the cache is partitioned into dedicated cache modules and configurable cache modules. The configurable cache modules can be reconfigured for computational usage, such as FIR, DCT/IDCT and encryption when an application does not need the whole cache.

2.3.4 Reduce static energy

Apart from the *dynamic power*, which arises from the capacitance charge and discharge on the output of hundreds of millions of gates in chips, static energy is increasing quickly as processor technology shrinks. For example, a 90-nm Pentium 4 consumes 110W, and static power takes roughly 40% of the total power dissipation. Static energy has been drawing increasingly more attention recently [57, 55, 85]. On-chip caches account for a significant portion leakage power of total power because of the large physical size. In order to decrease the leakage power of caches, various schemes [56, 78, 51, 54, 104] have been developed.

From the angle of circuit techniques, the approaches can be divided into two broad categories: **state-destructive** and **state-preserving**. State-destructive techniques gate the supply voltage, namely gated-Vdd, to reduce leakage in unused SRAM cells [78, 77, 51]. Gated-Vdd introduces a extra NMOS or PMOS Vdd-gated transistor. The gated-Vdd transistor is turned on to put the cell in "active" mode and turned off for the cell to be in "standby" mode. Turning off cache lines can save maximal energy. However, the loss of data can significantly induce significant energy overhead and performance overhead because of the additional cache miss caused. Instead of turning off cache lines, state-preserving approaches put the unused cache lines into low voltage/power state in which data can be retained. Thus the major advantage over the state-destructive is that the cost of wrongly putting a cache line to be accessed soon into low power mode is smaller. For example, drowsy

cache [56, 54] uses dynamic voltage scaling technique to select two supply voltages in each cache line for low power mode and normal mode. The waking up penalty of a cache line takes about 1 to 2 cycle and consumes little energy. This can avoid cache misses induced by data loss caused by shutting down cache lines and thus more energy savings can be obtained. One simple policy of controlling drowsy cache is to periodically put all cache lines into drowsy mode regardless of access patterns. The cache lines will be waken up when it is accessed later. This control strategy is **application-insensitive** which means that the cache lines is periodically turned off. In contrast to application-insensitive strategy, **application-sensitive** strategy manages the cache lines based on runtime performance feedback [77, 104].

2.3.5 Dynamic voltage scaling

The techniques introduced in the previous section disable functioning of unused resources to save energy. Apart from these methods, another very important class of approaches reduce the supplying voltage and slow down the speed of execution to decrease energy consumption using **dynamic voltage scaling**(DVS) [31, 45, 18, 23, 63, 11, 17, 39]. The dynamic power dissipation for CMOS circuits is proportional to the square of supplying voltage. Therefore, DVS techniques can effectively reduce CPU power by lowering the supply voltage. DVS techniques exploit the fact that the average computational throughput is often much lower than the peak computational capacity required. The processor runs at full speed at peak computation period to main performance and at low speed to meet the average computational requirement for energy savings. However, lowering supply voltage will significantly increase the latency of circuits and this tends to cause performance degradation. As a consequence, DVS techniques must consider the impact on the performance, power-latency product and energy-latency product which are among the most important research issues for DVS techniques.

For real-time applications, there are two kinds of slack time which can be exploited by DVS: worst-case slack time and workload-variation slack time [63]. The former one is the extra time when the processor utilization, which is computed based on WCETs(worst-case execution time) of tasks, is lower than 1. This means the processor can run slower to save energy without missing the deadlines of tasks in the worst case. The workload-variation slack happens when tasks are executed faster than WCET. The worst-case slack is exploited by [37, 45]. They assume that all tasks run at their WCETs and as a consequence, the workload-variation slack is not exploited. This is overcome by Shin and Choi [89] and the workload slack is exploited. However, this approach scales supply voltage on task-by-task level and cannot fully exploit workload-variation slack. In order to overcome this problem, several studies [63, 88, 11] perform DVS within a task. To achieve this, Lee and Sakurai [63] divide a task into several pieces and control the voltage on timeslot-by-timeslot basis inside the application instead of operating system.

Some researchers [39, 38] exploit the imbalance between different hardware components, such as CPU and memory activities. The hardware components which are not fully loaded can be slowed down to save energy without significant performance loss. Hsu et al [39, 38] scale down the speed of CPU in memory-bound applications. Such applications have a significant mismatch between the computation units and memory subsystem. Compiler is used to identify the imbalance, estimate the maximum possible energy savings and the performance impact, and make the final decisions on voltage scaling. Substantial energy savings can be obtained with a small performance penalty.

The DVS techniques have been proven practical and applied on some commercial processors, such as XScale [68] and crusoe processor [26] and on some academic design efforts [18, 63].

2.3.6 Scratchpad memory(SPM) for low power computing

Scratchpad memory

Apart from various cache architectures developed for performance improvement and energy reduction, *scratchpad memory* (SPM) [12] is a very important alternative of cache in embedded systems. In contrast to the cache where the data replacement is controlled by hardware, the scratchpad memory does not have tags and is controlled by applications. Using scratchpad memory has the following advantages. First, it consumes less hardware resources and energy compared to a cache since the scratchpad does not have tags. Second, the software controlled scratchpad is more flexible and is able to avoid many cache conflicts. It is shown that SPM have 34% smaller area and 40% lower power consumption than a 2-way associative cache of the same capacity [12]. Third, the access to SPM is more predictable and thus it has better real-time property which is desirable for real-time applications. Because of this, many lower power embedded processors and DSPs contain SPM, such as Motorola 68HC12BC32 [4], low power processor TMS320VC5501 from Texas Instruments [3], and low power processor lpc3180 from ARM company [65].

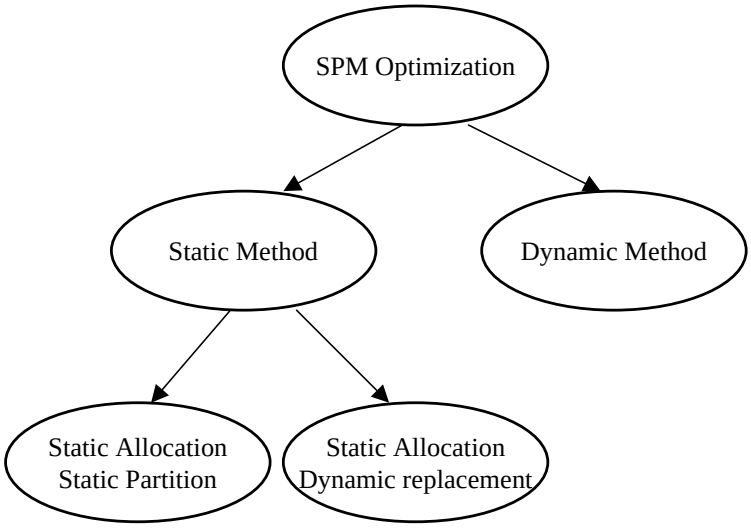

Figure 2.2: SPM optimizations

Since SPM is controlled by software or applications, intensive program analysis and carefully allocating codes/data into SPM are of crucial importance to achieve performance improvement and energy savings. A lot of work has been done on SPM. We can classify the existing SPM exploiting approaches into two categories: **static method** and **dynamic method**. Figure 2.2 shows the relation of different methods. Static method represents the class of approaches which statically allocate SPM resource and the SPM resource will not be changed during runtime. Static method can be further divided to static partition and dynamic replacement according to the policy of data replacement. Static partition method statically assigns the instructions or data into SPM and they will not be replaced during the program executions. For dynamic replacement method, the instructions or data can be replaced and the SPM can be used by multiple code blocks or data during runtime.

Static allocation, static partition

The method of static allocation and static partition has been studied in [7, 8, 94, 91, 92, 97, 73]. The methods of statically assigning instructions to SPM are introduced by [7, 8, 94]. Angiolini et al [7] develop a hardware fitting technique to partition the SPM resource for program segments and the address of SPM and main memory are overlapped. In this scheme, the hardware distinguishes the multiple address ranges of the program segments, and this requires a decoder with multiple comparators. As a result, no overhead of instructions such as jump is incurred while hardware penalty will be caused to map uncourageous ranges to SPM. This problem is modeled as SPM partitioning problem and can be solved in polynomial time using dynamic programming approach [7]. In contrast to the hardware fitting technique, software approach [8] uses the application to control the execution between SPM and main memory by inserting control transfer instructions such as jump instructions. The addresses of SPM and main memory are non-overlapped and can be accessed

independently.

The global and local data allocation to SPM for performance improvement is studied in [91, 92, 10]. The optimization problem is formulated as 0/1 integer linear programming (0/1 ILP) problem. The objective function (i.e. access time of memory access) and cost (i.e. size of SPM occupied), are identified, and optimal solution is found using ILP solver. Beside global and local variables, Avissar et al [10] also consider assigning stack variables to SPM for performance improvement.

Steinke et al [94] assign both instruction and data object to SPM for energy reduction. The compiler analyzes the application, and then the most frequently used instructions and data object are statically partitioned to SPM. The energy cost and savings for instructions and data partitioning are identified, and the optimal solution is found using Integer Linear Programming(ILP).

None of the work introduced above considers the existence of cache and the influence of SPM partitioning to cache is not considered. The instruction and data partitioning to SPM can affect the miss rate of cache and the change of cache miss rate can have significant impact on performance and energy consumption. To solve this problem, cache aware allocation approaches [97, 73] are developed which consider not only the energy savings by putting instruction or data object to SPM but also the effect of cache conflicts. Verma et al [97] partition instructions to SPM considering the insistence of cache. The cache behavior is modeled by a directed graph, namely conflict graph, whose nodes are instruction traces, and each edge between two nodes represents that one node can be replaced by the other. The total cache misses of a node are caused by all the other nodes that are not assigned to SPM. The cache misses of the node can thus be calculated from the conflict graph. Once having the expression of cache miss, cache hit, and SPM hit, the problem can be modeled as ILP problem and solved by ILP solver.

Static allocation, dynamic replacement

The methods introduced in previous section statically map hot blocks into SPM. These static schemes have the advantage of zero run-time copy overhead. Furthermore, global optimal placement can be achieved to maximize the effectiveness of SPM usage over the entire run of the application. However, the major disadvantage of static approaches is that the memory objects can not be replaced during execution. This may lead to under utilization of SPM when a program has multiple important loops which cannot fit into SPM. To overcome this problem, dynamic replacement approaches have been proposed [93, 98, 82]. These approaches use compiler to insert copy instructions into the program to replace instruction blocks in SPM and redirect the instruction fetch to the SPM during runtime. The optimal dynamic replacement decision of isntructions/data in SPM is accomplished using ILP [93, 98]. The ILP-based methods may not be practical in terms of runtime and often fail to scale to large size applications. To overcome this problem, Rivindran et al [82] propose an inter-procedural heuristic for identifying hot instruction traces and partitioning them to SPM. This heuristic based method is more practical for large program over ILP approaches although it does not achieve optimal results.

The dynamic replacement approaches introduced do not consider the existence of cache. Panda et al [73] consider the existence of cache and dynamically partition data to SPM with the goal of avoiding data conflict in cache for performance improvement. The lifetime distribution of variables is abstracted and analyzed to perform the partitioning.

Dynamic allocation

All the previous approaches of allocating SPM discussed above are static from a hardware perspective, i.e., the hardware parameters of SPM are not changed during

execution. As shown in previous sections, there are multiple phases within an execution which show different characteristics and various requirements to hardware parameters. Consequently, statically allocating SPM is not flexible enough to meet this requirement. In order to tackle this problem, an approach has been developed to dynamically allocate SPM [60]. The dynamic architecture, namely SCIMA [60], consists of four cache banks. Each of them can be dynamically configured as SPM or cache to achieve more flexibilities for performance improvement.

2.3.7 Summary of architectural methods

Two major classes of architectural methods for low power memory have been introduced in previous sections: designing and customizing cache architecture and applying SPM. The basic idea is to tune and manage memory architecture for a given application. In this thesis, we propose and design new reconfigurable cache architectures to tune SPM and cache parameters for energy reduction and performance improvement.

Cache architecture methods

Designing specific caches and customizing cache parameters can effectively reduce energy consumption. Pseudo set-associative caches first predictively examine one cache bank and only access the other banks if the initial access misses. This cache can reduce energy consumption if the hit rate of predictive accesses is high. Another cache variation, selective way cache, shuts down under-utilized cache banks to reduce energy consumption. All these cache variations try to disable under-utilized cache resources to reduce energy consumption. However, these caches have the drawback that the capability of under-utilized cache resources is wasted. In

order to exploit for more energy savings, our proposed PRIM takes advantage of the under-utilized resources instead of shutting them down. PRIM can configure a under-utilized cache bank and a normal active cache bank as a low power buffer pair where the voltage and frequency are halved. The throughput can be maintained by speculatively fetch instructions from these two cache banks. Through this method, the energy consumption of instruction memory can be reduced while performance can be maintained.

SPM methods

Apart from cache architecture methods, SPM methods are very important alternatives for energy reduction. Various methods are developed to partition instructions to SPM for decreasing energy consumption based on fixed architectural parameters. These methods can achieve significant energy savings, however they cannot change architectural parameters to suit a given application. Our two proposed schemes, SRIM and DRIM, can tune architectural parameters of SPM for a given application to reduce energy consumption and improve performance. For a given application, SRIM statically divides a given hardware resource budget into certain size of SPM and cache, and also partitions frequently used instructions to SPM allocated for reducing energy consumption and improving performance. Although SRIM can use hardware storage resources efficiently, it is not flexible in that the hardware parameters cannot be changed for different execution phases. To deal with this, a dynamic scheme, called DRIM, is proposed to dynamically reconfigure hardware parameters at runtime for energy reduction. Our schemes can tune hardware parameters of SPM to suit a given application and this is seldom addressed by existing works.

2.4 Application optimization

Apart from the architectural method to improve performance and reduce energy, application optimization is the other class of approaches to optimize for performance and energy. These approaches include loop transformations, instruction/data layout optimization, data in-place optimization, code compact etc. The compiler plays the key role in this method.

2.4.1 Loop transformations

Loop transformations, such as loop interchanging, loop tiling, loop merging, loop unrolling, etc [101, 70], are widely used to improve memory performance. These methods on managing loops can substantially change the characteristics of the programs. For example, loop transformations can change the spatial and the temporal locality of memory accesses. By applying loop unrolling, the number of instructions in the loop increases and this in turn results in more opportunities for ILP(instruction level parallelism). In DSTE [21] methodology, the global data flow optimization and loop transformations are heavily performed at the beginning steps of the framework. Global data flow optimization aims at optimizing the most costly DSTE factors and removing the bottleneck in data flow to memory hierarchy. This process has very big impact on the performance of the whole system. Loop transformations can also be used to reduce or remove the intermediate buffer. For example, by using loop merging, different sequential tasks can be interleaved and parallelized; thus the buffers between these tasks may possibly be reduced considerably or completely eliminated. Data reuse decision technique tries to exploit the temporal reuse of data. Techniques have been developed for applying reuse analysis and then distributing the data into the memory units in such a way that the frequently used data are put into the faster and smaller memory units.

2.4.2 Instruction/data layout optimization

Apart from the architectural method to decease cache conflict, an alternative approach, the instruction layout optimization, are developed to decrease the instruction cache misses by optimizing the instruction layout in the main memory [67, 75]. The idea is to place frequently executed code blocks of a program next to each other to reduce the chances of instruction cache conflict while increasing spatial locality within the program. Pettis and Hansen [75] use a "closest is best" strategy. The most frequent caller-callee pairs will be placed next to each other in order to decrease the cache conflict misses. This method is architectural independent. Hashemi et al [35] took into account of the cache size, cache line size, and procedure size to perform a color mapping of cache lines to procedures. They report performance improvement obtained over the previous architectural independent method. In addition to taking into the architecture information, Gloy et al [29] make use of the temporal ordering information that summarizes the interleaving of the procedures in a program trace to perform the instruction positioning. They report performance improvement over the previous two methods.

2.4.3 Code/data compression: reduce bandwidth requirement

The lower level memory hierarchy is usually larger and more energy expensive. Thus the access to SDRAM consumes much more energy and takes significantly longer time compared to the cache and the SPM. Because of this, reducing the accesses to main memory can greatly improve performance and reduce energy consumption. Other than the data flow optimization which can reduce main memory accesses, code/data compression is another very important method to decrease accesses to main memory [102, 16, 100, 24, 84, 22].

Code/data compression makes use of lossless data compression techniques to encode instructions/data into more compact form and a de-compressor will decompress the compact form of instructions/data and deliver them to CPU. Through this mechanism, the communication between CPU and memory can be reduced, and this results in energy savings. However, it has two more major constraints: first, it must be able to decompress a code/data block in a relative small block, as the code/data stream can be accessed at many possible entries. Second, the decompressor has to be small, fast, and energy efficient since the overhead caused by decompressor should not surpass the gain obtained. To achieve this, usually a hardware decompressor recovers the compressed instructions/data. ARM7TDMI [83] uses a set of 16-bit shorter instructions and these instructions can be used when the processor runs on Thumb mode. A decompressor will directly translate a 16-bit Thumb mode instruction to a 32-bit full mode one. Another method [102] assume that the total number of distinct instructions is limited, and thus the width of an instruction can be much reduced by encoding all distinct instructions as consecutive numbers. By using this method, the width of an instruction becomes $\lceil \log_2 N \rceil$ where N is the total number of distinct instructions appearing in the code, and memory bandwidth usage can be reduced. One drawback of this method is that the width of an instruction still can be significant if N gets large. To overcome this limitation, Benini et al [16] improve this method by only compressing the set of the most frequently used instructions. They observed that the some of instructions are much more frequently used than others and most of time spend on executing 256 distinct instructions. As a result, compressing most used instructions can effectively reduce communication between CPU and memory while minimal bit-width can be sustained. The decompressor for this method needs to be changed for distinguishing the compressed and uncompressed instructions.

The principle of code compression can be extended to data blocks with some

additional difficulties. Data compression techniques and hardware architectures have been introduced by Benini et al. [15].

Chapter 3

SRIM: Static Reconfigurable Instruction Memory

3.1 Motivation

The memory hierarchy is the main bottleneck in modern computer systems, due to the ever increasing gap between the speed of the processor and that of the memory. This problem becomes even worse in embedded systems, as designers not only need to consider performance, but also energy consumption. The memory hierarchy consumes a large amount of chip area and energy, which are precious resources in embedded systems.

Customizing the memory hierarchy [21, 74] for specific applications is an important way to fully exploit the limited resources to maximize the performance. Reconfigurable logic resources are being used for customizing the hardware platform for specific applications. In traditional hardware-software co-design methodologies, much of the work have focused on utilizing reconfigurable logic to partition the computation. However, utilizing reconfigurable logic in memory hierarchy design is seldom addressed.

The density and complexity of reconfigurable devices have been increasing dramatically. FPGAs available commercially now have up to millions of gates and integrate millions of bits of on-chip memory storage. With this large amount of resources available, it is possible to design an embedded system using single FPGA chip. Software-core processors have been developed for FPGAs, such as the Altera Nios II [2] and the Xilinx MicroBlaze [1].

With the abundance of on-chip *Block Memory* (BRAM) resources, it is very important to make full use of these resources to improve the system performance. However, making use of reconfigurable resources for improving the performance of the memory hierarchy is seldom addressed. Our work focuses on exploiting reconfigurable logic to improve the performance of the instruction memory hierarchy for specific applications. We propose a reconfigurable instruction memory hierarchy consisting of an instruction cache and a scratchpad memory.

The concept of *scratchpad memory* (SPM) [12] is an important consideration in embedded systems. In contrast to the cache where the data replacement is controlled by hardware, the scratchpad memory is controlled by applications. Using scratchpad memory has the following advantages. First, the software controlled scratchpad is more flexible and is able to avoid many cache conflicts. Second, since the scratchpad does not have tag, it consumes less hardware resource and energy compared to a cache.

Given a fixed amount of reconfigurable on-chip logic resources and a specific application, we address the problem of partitioning the available resources into an instruction cache and a scratchpad, whose sizes depend on the application. Our goal is to lower the instruction fetch miss rate and improve the system performance, as well as conserve energy.

In this chapter, we propose a reconfigurable instruction memory hierarchy for

FPGA-based systems and an algorithm to explore the design space as well as perform instruction partitioning. To our knowledge, the design space exploration problem of partitioning a fixed amount of reconfigurable resources into an instruction memory hierarchy consisting of a cache and a scratchpad has not been addressed in the literature. Second, we have completed a prototype implementation of the proposed instruction memory hierarchy using actual FPGA hardware. Our implementation is based on the Altera Nios II platform.

We evaluated the performance of our proposed instruction memory hierarchy compared to one that consists of only an instruction cache. Using five benchmarks from the MediaBench and MiBench suites as workload, the experimental results show that our architecture provides significant performance improvements and energy reduction.

3.2 Related work

The rationale and benefits of reconfigurable cache memory architectures have been well studied by previous researchers [80, 103]. There are also previous work to study the benefits of the the scratchpad memory (SPM) and to develop techniques to manage the SPM.

Several researchers [97, 8, 94, 73, 49] designed algorithms to partition instructions or data into the SPM, with the goal of reducing the conflict misses and the energy consumption. However, most of them assumed that the memory hierarchy has a fixed-size cache and a SPM, and they did not optimize the architectural parameters of the memory hierarchy. For example, Panda [73] partitioned the data objects in an application into the SPM and the cache to reduce the amount of cache conflicts. But he only considered two fixed memory hierarchy configurations; namely, a

2 KByte cache compared with a combination of 1 KByte cache plus 1 KByte SPM. Kandemir's algorithm [49] can tune the size of the SPM, but it was designed for data references instead of instruction accesses. Vander et al [25] focused on adjusting the size of the instruction loop buffer for specific applications to maximize the energy savings. However, the cache is fixed and the problem they studied is different from ours.

Apart from partitioning instructions into the SPM, a complementary technique to optimize the performance of the instruction memory hierarchy is code positioning and mapping [75, 35, 29]. These techniques were developed to decrease the instruction cache miss rate by repositioning instruction blocks or procedures in the main memory.

Our work differs from the previous works and is novel in two important ways. First, we focus not only on partitioning the instructions into the SPM, but also on tuning the parameters of the memory hierarchy for specific applications. For a given amount of hardware resource budget for the instruction memory hierarchy, our algorithm partitions it into an instruction cache and a SPM. Our algorithm also assigns the instructions into the SPM by analyzing the instruction access characteristics of the specific application.

Second, unlike previous work which rely mainly on simulations to evaluate the techniques to improve performance in reconfigurable architectures, we have actually implemented our reconfigurable instruction memory hierarchy using real FPGA hardware. Our implementation, which is based on the Altera Nios II platform, enables us to study the actual performance impact of our technique on a real system.

3.3 Design flow and hardware architecture

3.3.1 Design flow

The design flow for our reconfigurable memory hierarchy is shown in Figure 3.1. The inputs are the application and the hardware resource budget for the instruction memory hierarchy. The outputs are the transformed binary code, and the parameterized instruction memory hierarchy consisting of a SPM and an instruction cache of a certain size.

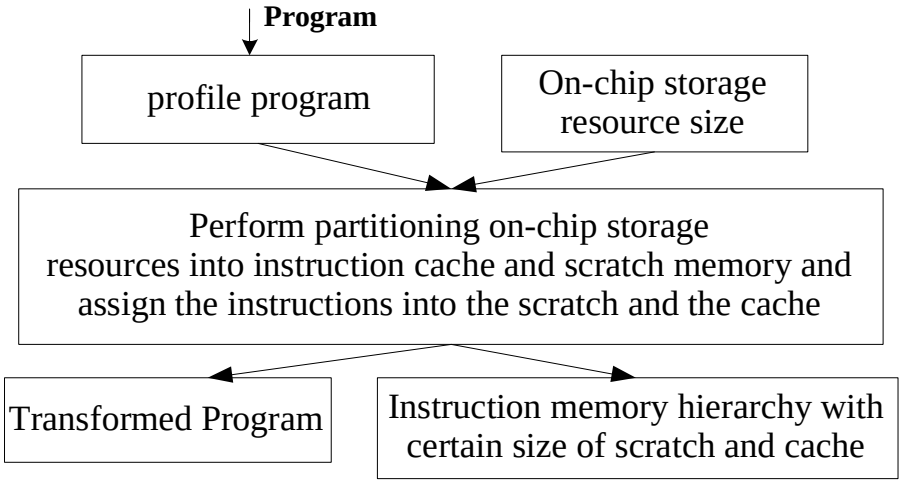

Figure 3.1: Design flow for the reconfigurable instruction memory hierarchy

The design process is as follows. First, we profile the application using the gcc compiler. Then, we compile the application into assembly code using the gcc cross compiler for the Nios II processor. From the assembly code, we build the data structure representations for our algorithm. These are then analyzed to determine the sizes of the SPM and the cache, as well as to partition the instructions into the SPM and the cache. We then patch the code that are allocated to the SPM to get the transformed assembly code.

The transformed assembly code will be compiled using the gcc cross compiler for the Nios II to generate the binary executable. The parameterized instruction

memory hieararchy coupled with the Nios II based system will be compiled into FPGA using the Altera QuartusII tool.

3.3.2 Instruction memory hierarchy architecture

The architecture of our reconfigurable memory hierarchy is shown in Figure 3.2. The memory hierarchy consists of an instruction cache, a SPM, a SPM controller (SPMC), an address lookup function unit (ALFU) and some other logic components.

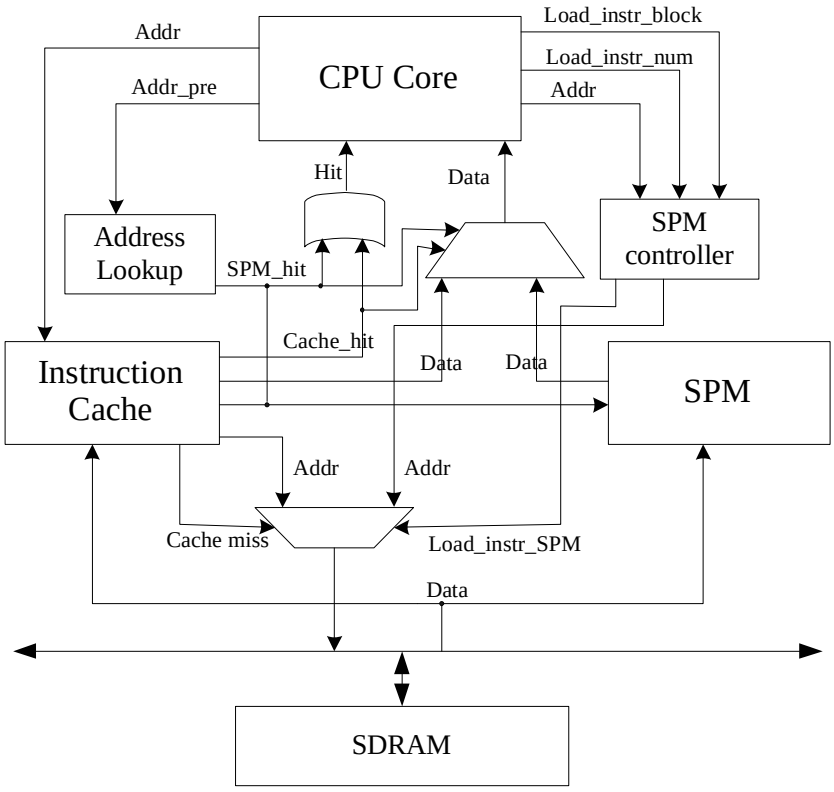

Figure 3.2: The architecture of the reconfigurable instruction memory hierarchy

The ALFU is responsible for deciding whether an instruction resides in the SPM or not. It consists of two address registers and two parallel comparators. The two registers store the upperbound and the lowerbound of addresses for the instruction block to be stored in the SPM. The highest instruction address of the block is stored in the upperbound register, while the lowest instruction address of the block is stored

in the lowerbound register. If the address of an instruction to be fetched falls within the range of these two registers, the ALFU will generate the signal SPM_hit, which is used to control the selection and the switching of the cache and the SPM.

The SPMC is used to load the instructions from the main memory into the SPM, and updating the values of the upperbound and lowerbound registers. Since the instructions are statically assigned into the SPM, the SPMC is actually optional. One can remove the SPMC from the memory hierarchy to save the resource usage.

The instruction memory hierarchy is integrated in the Nios II processor. It is implemented in the form of a Verilog description. The sizes of the cache and the SPM are parameterized, and the parameters can be easily changed in the Verilog source code.

3.3.3 The implementation of SRIM

To evaluate SRIM, we prototyped it on an FPGA board and collected the execution statistics from the board. We chose the Nios II Development Kit,Stratix Professional Edition from Altera company [2] as our implementation platform. The system consists of a Nios II processor, SRIM, and main memory (an off-chip SDRAM). The Nios II processor is soft-core RISC processor developed by Altera which is widely used in embedded systems. The Nios II development kit comes with a tool called SOPC which is used to build and configure the system. The users specify the parameters and configuration through SOPC and it will automatically generate the source code written in hardware description language, such as verilog or VHDL. The Quartus II tool takes the Verilog or VHDL source code as input and synthesize it into a configuration file for the FPGA residing on the board.

To implement SRIM, we first generate a standard Nios II based system using

SOPC. The automatically generated system mainly comprises of a Nios II process, a direct-mapped instruction cache and data cache, SDRAM controller, and components such as the one used to communicate with PC. After this step, we manually modify the Verilog source code and insert our SRIM component, also written in Verilog, into the system. In addition, we also added components to monitor the execution so as to collect execution statistics. These are counters triggered by signals of the hardware. The statistics we collect are follows:

1. *#Exe-cyc*: the number of clock cycles an execution took. The counter is triggered by the main clock.

2. *#I-fetches*: the number of total dynamic instruction fetches that the processor made. This counter is triggered by the read signal to the instruction memory hierarchy.

3. *#I-misses*: the number of dynamic instruction fetch misses. The counter is triggered by the instruction hierarchy miss signal.

Our two metrics used for comparison are the execution performance and the dynamic energy consumption of the instruction memory hierarchy. The execution performance can be obtained directly from the *#Exe-cyc*, while the dynamic energy consumption is calculated from *#I-fetches* and *#I-misses*.

3.4 Design space exploration and instruction partitioning

A key aspect of our proposed instruction memory hierarchy is our algorithm for design space exploration and instruction partitioning. The design space exploration is

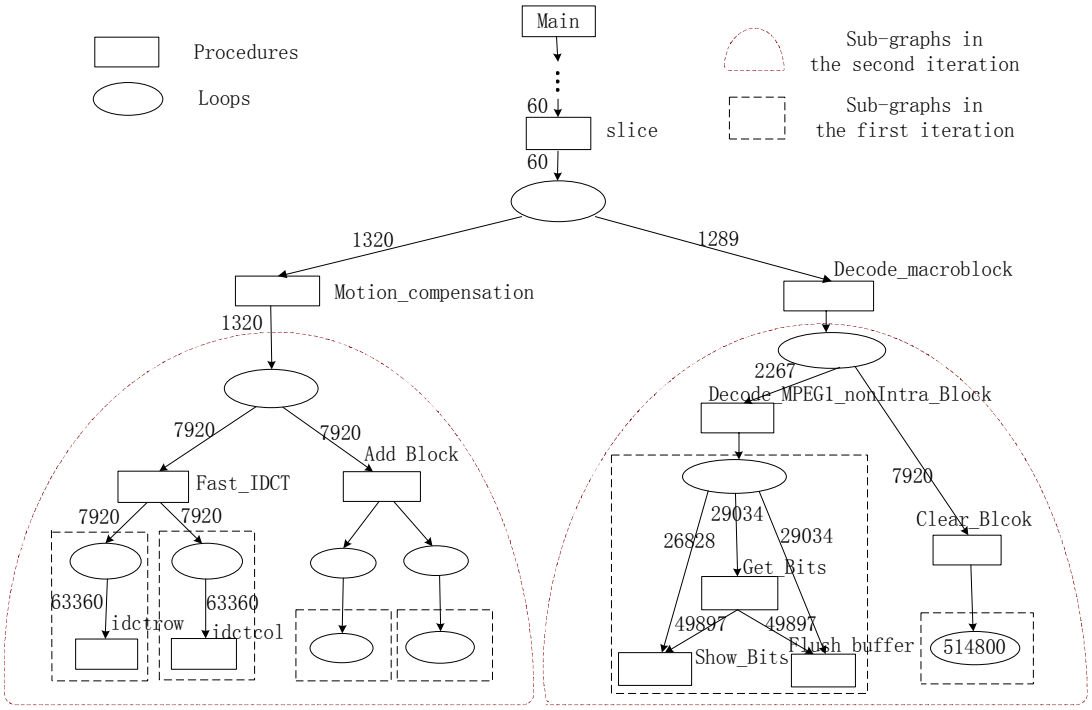

Figure 3.3: The loop-procedure hierarchy graph from mpeg2decode

to determine the size of the SPM and the instruction cache for a specific application, while the instruction partitioning is to assign the instructions into the SPM and the cache to maximize the performance and minimize the energy consumption.

3.4.1 Loop-procedure hierarchy graph

Our algorithm uses the Loop-Procedure Hierarchy Graph (LPHG) [64] to represent a program. In Figure 3.3, we show the LPHG representation of a key portion of the mpeg2decode benchmark from the MediaBench suite. For simplicity, not all of the procedures and the loops in mpeg2decode are shown. In this graph, the rectangular nodes represent the procedures, while the ellipse nodes represent the loops. The arrows or edges between the nodes denote the relationship between them. For convenience, we call the node pointed to by an arrow a successor, while the node that points to another node is a predecessor. The numerical value associated with each arrow is the number times the successor is entered from the predecessor.

3.4.2 Proposed algorithm

Our proposed algorithm is shown in Algorithm 1. We assume that most of the instruction cache conflicts are caused by the loops. The intuition is to put as much of the most frequently executed loop kernels into the SPM so as to reduce the instruction cache conflict misses within each kernel. Starting with a storage resource of size M to be partitioned into the cache and the SPM, we divide this partitionable storage into two portions of size $M/2$ each. We will try to fill up one of these portions with instructions for the SPM. Then, we recursively partition the other portion until the size of the remaining storage is so small that if it were to be further partitioned, there will be a large amount of cache conflicts when executing a kernel.

Definition 1: a subgraph spanned by the loop L, denoted by *SubG(L)*, is the subgraph consisting of all the nodes that are reachable from the loop node L. The size of a *SubG(L)*, denoted by *Sizeof(SubG(L))*, is the number of instructions in the *SubG(L)* whose execution frequency in the loop exceed a certain threshold value, denoted as $ExFreqTh$.

The idea is that within the *SubG(L)*, there are certain instructions that are executed more frequently than others, and will be considered for placement in the SPM. In this work, we set the $ExFreqTh$ value to be 1/3 of the number of iterations in loop L.

The algorithm starts from the leaf loops of the LHPG and work upwards during the course of its execution. This is because the deeper the loops are in the LPHG, the more frequently their instructions are executed. So, these loops should be assigned to the SPM. However, as we move to the upper levels of the LHPG, two or more *SubGs* at the level might be in conflict.

Definition 2: *SubG(L1)* and *SubG(L2)* are in *potential conflict* if *L1* and *L2*

are not a successor of each other. Given a partitionable storage of size M, *SubG(L1)* and *SubG(L2)* are in *conflict* if

- *SubG(L1)* and *SubG(L2)* are in potential conflict
- *Sizeof(SubG(L1))* $>$ *M/2* and *Sizeof(SubG(L2))* $>$ *M/2*

At a particular level of *SubGs*, if no pair of *SubGs* are in conflict, then half the partitionable storage should be added to the SPM. The instructions are then assigned to the SPM according to a calculated heuristic value. This process is repeated recursively with the remaining storage as the algorithm moves up the LHPG, until at least two *SubGs* at a particular level are in conflict.

The instruction partitioning granularity is at the loop and procedure level. As shown in line 4 of Algorithm 1, in a *SubG*, the frequently executed instructions in the loops and procedures will be selected to form the instruction blocks. The criteria to select the instructions is that the execution frequency of the instructions should be larger than $ExFreqTh$.

The heuristic value for the selection of instruction blocks is computed in line 5 of the Algorithm 1. In this work, we set the heuristic value as: $V_h = E_{avg} + 5 * Number\_of\_times\_called$. The E_{avg} represents the average instruction execution frequency of a selected block, while the $Number\_of\_times\_called$ stands for the number of the times the procedure containing the instruction block is called. Next, in line 6, the unimportant *SubGs* will be pruned from the list of *SubG*. This occurs if the E_{avg} of a *SubG* is less than 1/20 of the average execution frequency of all the select instruction blocks. Then we regard it to be unimportant and will not consider it in the current iteration.

In our algorithm, the procedure $two\_SubGs\_conflict$ is used to determine whether there exists two *SubGs* in the list of *SubGs* which are in conflict. If two *SubGs* are

Algorithm 1: Design space exploration and instruction partitioning

Input: Assembly code and source code profiling information,storage size
Output: The SPM size and the assignment of instructions into SPM
Variable PM : current partitionable storage size;
1 Build Loop-Procedure Hierarchy Graph(LPHG)
2 $PM = M$;
3 **while** *($PM >$ Lower_Bound_of_Cache_Size)* **do**
4 Select the instruction blocks for loops and procedures in SubGs list;
5 Calculate the heuristic value, V_h, for all the selected instruction blocks;
6 Prune the non-important SubGs;
if *!two_SubGs_conflict(SubGs list, PM)* **then**
Select instruction blocks from SubGs until $PM/2$ amount of storage is filled;
$PM = PM/2$;
goto Update;
else //if there is a conflict between at least two SubGs.;
Break;
// break the while loop Update: Update SubGs list;
end
Cache size = PM, Scratch size = M - Cache Size;
Pack all the extracted instruction blocks into the determined size of SPM;
return;
7 **Procedure two_SubGs_conflict(SubGs_list, PM)if** *(the size of all SubGs in the SubGs list is smaller than PM/2)* **then**
return false
else if *(size of only one graph in the SubGs_list is larger than PM/2)* **then**
return two_SubGs_conflict(list of all the child SubGs in the current SubG, PM);
end
else
return true;
end

in conflict, which means their sizes are larger than $PM/2$, further partitioning the current partitionable storage into the SPM and the cache is likely to cause more cache conflicts. Thus, we stop the partitioning process.

On the other hand, if none of the *SubGs* are in conflict, we can safely assign $PM/2$ amount of storage to the SPM. Then, we select the instruction blocks from the *SubGs* to fill the $PM/2$ amount of storage in the SPM according to the heuristic value, V_h computed earlier. Once the SPM is filled, the list of *SubGs* will be updated and the process repeats.

Apart from the detection of conflicts between *SubGs*, another termination criteria of the algorithm is that the remaining storage for the cache should not be lower than a threshold value, $Lower\_Bound\_of\_Cache\_Size$.

In the mpeg2decode example in Figure 3.3, the *SubG_list* initially contains all the leaf *SubGs* in the LPHG. From the figure, all the *SubGs* in dashed boxes are in the *SubGs_list* of the first iteration, while the *SubGs* in the hemispheres are in the *SubGs_list* of the second iteration. For mpeg2decode, the given on-chip storage is 2048 bytes. The size of every *SubG* in the first iteration is less than 1024 bytes. Thus we can partition the half the storage into the SPM. After assigning the instructions into the SPM, the *SubGs_list* will be updated and the second iteration starts. However, the algorithm terminates at the second iteration because the size of both the SubGs (shown in the hemispheres in Figure 3.3) in the *SubGs_list* are larger than 512 bytes. Thus, the final memory hierarchy configuration consists of 1024 bytes of instruction cache and 1024 bytes of SPM. The instruction assignment has been completed during the first iteration.

3.4.3 Code transformation for SRIM

In SRIM, the application loads instructions into the SPM from the main memory at the beginning of the execution. For simplicity, the instructions from the different segments are merged into a large block to be load into SPM before the execution. We have added a custom instruction, `SPM_load` (load instruction block to SPM), to the Nios II instruction set for this purpose. The SPM load instruction will load the instructions immediately following it. The number of instructions to be loaded is specified as an argument of SPM load.

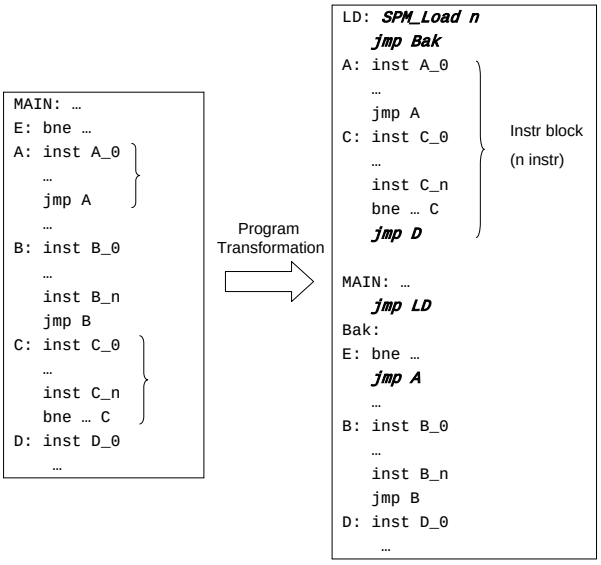

Figure 3.4: Code transformation for SRIM

After identifying the instructions to be placed in the SPM, we will gather them together to form a single instruction block to be loaded to SPM by the `SPM_load` instruction. The issue that needs to be solved is the preservation of the original control flow. In order to do this, we have to insert new jump instructions and/or delete instructions. Figure 3.4 shows an example. The instruction blocks `A` and `C` are moved and grouped together to form a single instruction block to be loaded to SPM. The left side of the figure shows the original source code, while the right side is the transformed code. For instruction block `A`, we add a `jmp A` at location `E`. This

is because the instruction immediately before block `A` is a conditional jump. Thus there is a control flow from block `E` to block `A`. To preserve this flow, an absolute jump has to be inserted. For block `C`, a jump instruction `jmp D` is inserted to the end of block `C` to retain control flow from `C` to `D`. After grouping the instructions together and inserting jump instruction to retain the control logic, we insert a '`SPM_load n`' instruction used to load the merged block into the SPM, where '`n`' specifies the number of instructions to be loaded.

In the last step, a jump instruction is added at the entry of the program, denoted by `MAIN` in Figure 3.4. This instruction subsequently causes the loading of the instruction block into the SPM at the beginning of the execution.

3.5 Performance evaluation

3.5.1 Experimental methodology

We used the Nios II Development Kit (Stratix Edition) [2] as our development platform. We implemented the reconfigurable memory hierarchy shown in Figure 3.2 by modifying the Nios II Verilog source code. The cache is direct-mapped, with line size of 32 bytes. Hardware counters are added to the system to profile the program execution statistics, such as the number of cache misses, the number of instruction fetches, the number of issued instructions, and the total number of cycles taken. We then synthesized the system using the QuartusII tool.

In our experiments, we used five application benchmarks from the MediaBench and the MiBench suites. We compared the performance and the energy consumption of executing each benchmark on two different instruction memory hierarchies: (1) a

baseline instruction memory hierarchy with only an instruction cache, and (2) our instruction memory hierarchy combining a smaller instruction cache and a SPM.

We execute these benchmarks and collect the performance statistics from the hardware counters. Since we cannot measure the energy consumption directly from the hardware platform, we model the the energy consumption using the CACTI [99] model for 0.5 μm technology. In the calculation of the energy consumption of our instruction memory hierarchy, we included the logic elements connecting the instruction cache and the SPM to the Nios II processor. However, we exclude the SPMC from the energy consumption calculation, since it is only used to load the instructions into the SPM at the beginning of the application execution and it is actually an optional component. We assume that the row activation and precharge of the SDRAM consumes 20nJ, while column access consumes 26nJ [33].

3.5.2 Performance improvements and energy savings

Performance: The performance results are shown in Table 3.1. N_f(K) represents the total number of instructions fetched (in thousands), and $R_{SPM}(\%)$ stands for the percentage of the number of instruction fetches from the SPM out of the total number of instruction fetches. T_{exe}(sec) represents the application execution time in seconds.

From the results, the decrease in the instruction fetch miss rate for the benchmarks studied ranges from 6.3% to 69.7%. The average improvement in the miss rate is 33.2%. This improvement in the instruction fetch miss rate for our instruction memory hierarchy over the baseline cache configuration comes from the mapping of the frequently executed instructions into the SPM.

As a result of the improvement in miss rate, the execution times of the ap-

Table 3.1: Performance results

Benchmark	Configuration	N_f(K)	R_{SPM}(%)	miss rate(%)	T_{exe}(sec)
g721encode	2048 IC	195,124	0	3.09	6.14
	512 IC+1524 SPM	196,057	82.2	1.93	5.45
Improvement	–	–	–	37.5%	11.2%
g721decode	2048 IC	293,637	0	2.51	9.16
	512 IC+1508 SPM	299,005	86.0	1.35	8.15
Improvement	–	–	–	46.2%	11.0%
Dijkstra	256 IC	48,582	0	2.06	2.24
	64 IC+192 SPM	49,789	73.2	1.93	2.24
Improvement	–	–	–	6.3%	0.0%
Blowfish	2048 IC	25,617	0	2.21	0.85
	64 IC+1984 SPM	25,621	94.6	0.67	0.74
Improvement	–	–	–	69.7%	12.9%
mpeg2decode	2048 IC	38,532	0	1.44	1.46
	1024 IC+1024 SPM	38,567	51.9	1.35	1.45
Improvement	–	–	–	6.3%	0.7%
Avg Improv	–	–	–	33.2%	7.2%

plications are decreased by an average of 7.18% for the benchmarks studied. The improvement in execution time is not as impressive as the improvement in miss rate. A possible reason is that because of the low clock frequency ($50MHz$) of the hardware platform, the cache miss penalty becomes less important.

Energy consumption: The energy consumption results are shown in Table 3.2. E_{ic}(nJ) represents the energy consumption per access to the instruction cache, while E_{SPM}(nJ) represents the energy consumption per access to the SPM. The total energy consumption during the execution of the benchmark for the two instruction memory hierarchies are computed, as well as the reduction in the energy consumption due to our instruction memory hierarchy.

From the results, the reduction in the energy consumption for the benchmarks studied ranges from 8.0% to 56.9%. The average reduction in the energy consumption is 30.1%. A major contribution to the energy reduction is the decrease in miss rate since SDRAM accesses consume a lot of energy. The other factor that causes the energy reduction is the lower energy consumption per access to the SPM compared

Table 3.2: Energy consumption

Benchmark	Configuration	E_{ic}(nJ)	E_{ic}(nJ)	E_{SPM}(nJ)	Energy(μJ)	Reduction(%)
g721encode	2048 IC	1.51	–	–	1,669,325	
	512 IC+1526 SPM	–	1.33	1.12	1,089,642	34.7%
g721decode	2048 IC	1.51	–	–	2,123,818	
	512 IC+1526 SPM	–	1.33	1.12	1,264,014	40.5%
Dijkstra	256 IC	1.09	–	–	281,134	
	64 IC+192 SPM	–	1.10	0.83	258,543	8.0%
Blowfish	2048 IC	1.51	–	–	161,386	
	64 IC+1984 SPM	–	1.10	1.19	69,534	56.9%
mpeg2decode	2048 IC	1.51	–	–	184,691	
	1024 IC+1024 SPM	–	1.41	1.04	165,683	10.3%
Avg Reduct	–	–	–	–	–	30.1%

to that of the cache, and most of the instruction fetches are from the SPM.

Resource usage: Two different types of resources in FPGA are used to measure the resource usage, the logic array blocks (LABs) and the block RAM of size 512 bytes (BRAM512). The data is obtained from the place-and-route report of Quartus II. The number of LABs reported is the total amount of LABs used by the entire platform, while the BRAMs reported are only those used by the instruction memory hierarchy. For a pure 2Kbytes instruction cache, 533 LABs and 35 BRAM512 are used, while for SRIM consisting of 1Kbytes of cache and 1Kbytes of SPM, 555 LABs and 34 BRAM512 are used. One BRAM512 is saved by the proposed memory hierarchy. One BRAM512 is saved because less tag storage is needed. However, the proposed memory hieararchy consumes 22 more LABs, most of which is consumed by the SPM controller.

3.6 Summary

The performance and energy consumption of instruction memory hierarchy is of crucial importance for embedded systems. SPM is a very important alternative

of cache which can be used for improving performance and reducing energy consumption. The SPM parameters of previous methods are fixed and this is not flexible as different applications have different hardware requirement. Toward this, we proposed and designed a static reconfigurable instruction memory hierarchy for embedded systems. Given a fixed amount of on-chip storage and a specific application, our algorithm partitions the storage into an instruction cache and a SPM, and assigns instructions to the SPM and the cache for maximizing the performance and reducing the energy consumption.

We believe that our approach is novel because previous work are not as flexible in their design space exploration of the instruction memory hierarchy. Furthermore, we have implemented the proposed instruction memory hierarchy using FPGAs and obtained actual system performance results, instead of using simulations for the performance evaluation. Our results show that the proposed architecture can achieve a significant performance improvement and energy reduction.

Chapter 4

Integrated memory hierarchy design

4.1 Motivation and related work

In the previous chapter, we propose and design a reconfigurable instruction memory architecture. The idea is to partition the memory storage resource of the embedded processor into an instruction cache and a scratchpad memory (SPM) [12] according to the characteristics of the given application. By analyzing the given application, the storage resource will be partitioned into a SPM and a cache of a certain size to suit the application. Instructions will be assigned into the SPM statically, in order to decrease the instruction cache miss rate as well as to reduce the energy consumption of the overall memory hierarchy.

Traditionally, there are two main methods to improve the instruction memory hierarchy performance. The first method is instruction layout optimization, and the second method is the architecture exploration to determine the optimal instruction memory hierarchy for a particular application. The basic idea of instruction layout

optimization [75, 35, 29] is to place frequently used sections of a program next to each other in the address space. In this way, instruction locality will be improved and conflicts in the instruction cache will be reduced. In memory hierarchy architecture exploration, the instruction memory hierarchy is customized for a specific application [21, 74], and it can be statically parameterized or dynamically reconfigured according to the characteristics and demands of the application. Our SRIM belongs to the class of methods of architecture exploration.

Although these two methods are well established, the interaction between them is seldom studied. System designers either perform instruction layout optimization based on fixed memory hierarchy, or tune the memory hierarchy without considering instruction layout optimization. In this chapter, we propose a framework that combines these two methods to further improve the performance of the instruction memory hierarchy in embedded systems.

During the instruction partitioning process for SRIM, it is assumed that the reduced cache can handle all of the remaining frequently executed loops that are not placed in the SPM. However, even if the cache's size is larger than the sizes of the loops, the layout of the instructions may not be ideal. Some frequently executed instructions of the loops may be mapped into the same cache line, and cause severe cache conflicts. We can alleviate this problem by optimizing the layout of the instructions that are to be cached.

Similarly, the instruction layout can be made more effective by tuning the instruction memory hierarchy. For example, if an application contains only one loop and the cache's size is smaller than the loop's size, then even the optimal instruction layout optimization method cannot result in good performance. In this case, we should allocate as much storage resource as possible to the SPM and try to relocate the bulk of the loop in the SPM.

To combine the instruction memory hierarchy exploration and the instruction layout optimization effectively is non-trivial since they affect each other. Architecture exploration and instruction partitioning can change the conflict relationships between different instruction blocks. Thus, to achieve a better solution, our framework considers this factor when building the instruction conflict metrics for instruction layout optimization. Furthermore, since these two methods are inter-dependent, the phase order in which they are applied may affect the overall performance of the instruction memory hierarchy. We will study this and other issues in this chapter.

We have developed a prototype of the framework in a compiler to perform the memory hierarchy exploration and instruction layout optimization. Our experimental results using five benchmarks from the Mediabench and the MiBench suites show that our framework can achieve significant performance improvements and energy reduction over the traditional methods.

Related work Instruction layout optimization techniques have been developed to decrease the instruction cache miss rate by changing the way instructions are laid out in memory. Pettis and Hansen [75] used a "closest is best" strategy in which the most frequent caller-callee pairs are placed next to each other in order to decrease the cache conflict miss. Their method is independent of the underlying architecture. Hashemi et al [35] took into account the cache's size, cache line size, and the size of the procedures to perform a coloring-based mapping of cache lines to procedures. They reported a performance improvement over the previous architecture independent method. Apart from considering architecture information, Gloy et al [29] also made use of temporal ordering information that summarizes the interleaving of procedures in a program trace to perform instruction layout. They reported a further performance improvement over the previous two methods. These three methods are based on a fixed memory hierarchy. They do not tune the instruction memory hier-

archy configuration to make better use of the instruction layout optimization, and do not perform the layout optimization which considers the possibility of alternative configurations of the memory hierarchy.

Apart from the instruction layout optimization techniques, another promising approach is the application-specific tuning of the memory hierarchy. Several researchers [80, 103, 95, 14] designed dynamically reconfigurable memory hierarchy to meet the requirements of the applications. Others performed memory hierarchy exploration statically for specific applications [27, 72, 49]. Jain et al. [46] and Huang et al. [41] proposed the integration of memory hierarchies and computation units to improve the performance. Vander et al. [5] performed the exploration for the instruction loop buffer configurations for specific applications and mapped the loops into loop buffers to reduce the energy consumption. The main drawback of all these methods is that they did not consider the instruction layout optimization when tuning the memory hierarchy.

The main contribution of our work is that our framework combines instruction layout optimization and memory hierarchy exploration effectively to improve the overall instruction memory hierarchy performance. To the best of our knowledge, this is the first work that combines the software optimization of instruction layout with instruction memory hierarchy exploration, and assesses the performance impact of combining these two approaches.

4.2 Framework design

The structure of our framework is shown in Figure 4.1. The inputs to the framework are the given application and the storage resource budget for the instruction memory hierarchy. The outputs are the partitioned instruction memory hierarchy for the

application, and the transformed application with the optimized instruction layout. The framework consists of several major steps:

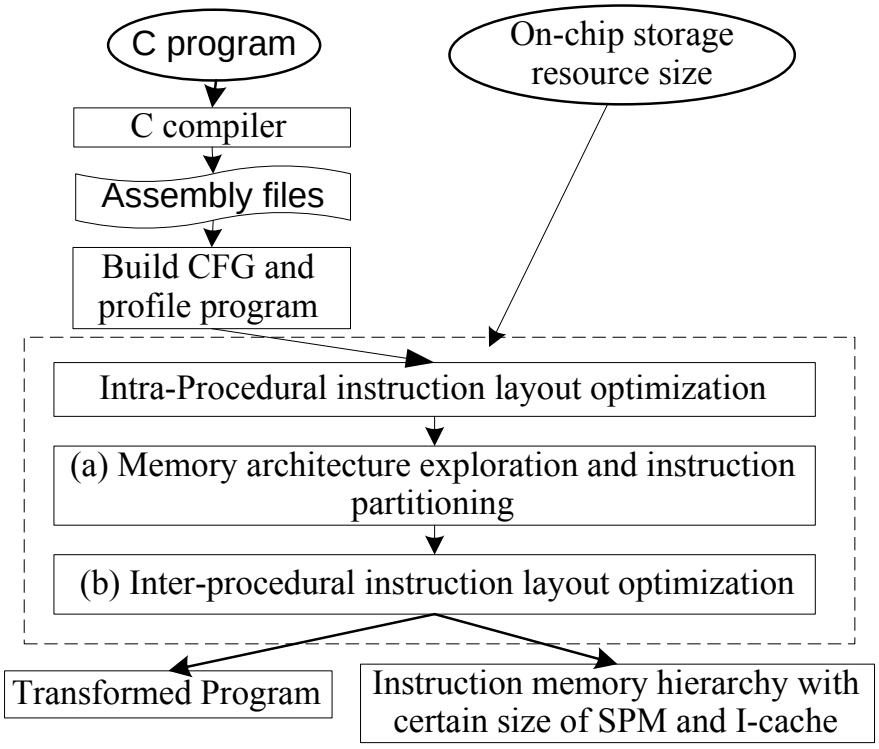

Figure 4.1: Our integrated framework.

- *Profiling the application:* In this first step of the framework, we obtain the characteristics of the application's dynamic execution by profiling. The data we collect include the execution counts of the edges of the control flow graphs (CFGs) of all the procedures and the number of the procedure invocations. We achieve this by building the CFG for each procedure and then add instructions to instrument each basic block of a CFG. The instrumented program is then executed to get the execution statistics.

- *Intra-procedural instruction layout optimization:* The goal of this step is to optimize the instruction layout within each procedure according to the profiling statistics obtained in the previous step.

- *Division of the storage resource and instruction partitioning:* In this step, we analyze the application's characteristics and decide on the suitable division of the storage resource budget into a SPM and an instruction cache. Once this is done, we will relocate frequently executed instructions into the SPM.

- *Inter-procedural instruction layout optimization:* This step is to optimize the instruction layout across the different procedures. Conflicts can be further decreased by applying this optimization.

We have developed a prototype of the framework based on the Simplescalar tool set consisting of a gcc-based compilation system and a simulator. First, we modified the Simplescalar simulator such that it can support our own instruction memory hierarchy consisting of a SPM and an instruction cache. Second, we built an instruction optimization tool which performs the program profiling, inter-procedural and inter-procedural instruction layout optimization. We adopted the memory hierarchy exploration method proposed in our earlier work [27].

4.3 Instruction layout optimization

A major component of our framework is the instruction layout optimization tool. The main functionalities of this tool are to profile the application, and to perform intra-procedural and inter-procedural instruction layout optimizations. Our implementation of both the intra-procedural and inter-procedural layout optimizations are based on existing established algorithms.

Algorithm 2: Intra-procedural instruction layout optimization.

Input: *Proc_list*: The list of all procedures of the application after the intra-procedural optimization
Input: *PS*: the profiling statistics: *PS*
Output: *Proc_list_opt*: Procedure list whose procedures have been intra-procedural optimized

```
1 foreach p in Proc_list do
2     perform intra-procedural optimization for p using PS;
  end
3 return Proc_list;
```

4.3.1 Intra-procedural instruction layout optimization

Algorithm 2 shows our intra-procedural layout optimization. We make use of the Top-down Positioning algorithm [75] for the intra-procedural instruction layout optimization, as shown in Line 2 of Algorithm 2. The algorithm first places the entry basic block for the procedure. Then, the successor with the largest execution count will be selected if it has not already been placed yet. If all the successors have been placed, then the basic block with the largest connection to the already placed basic blocks will be chosen. This algorithm continues until all the basic blocks have been placed. Figure 4.2 shows an example of how the intra-procedural optimization works. Figure 4.2a shows the CFG of a procedure, while Figure 4.2b shows the original layout of the basic blocks in memory. The instruction layout after the intra-procedural optimization is shown in Figure 4.2c.

It is useful to perform the intra-procedural optimization. First, it can improve the instruction locality. Second, the code may become more compact due to the reordering of the basic blocks. Finally, procedure splitting becomes feasible.

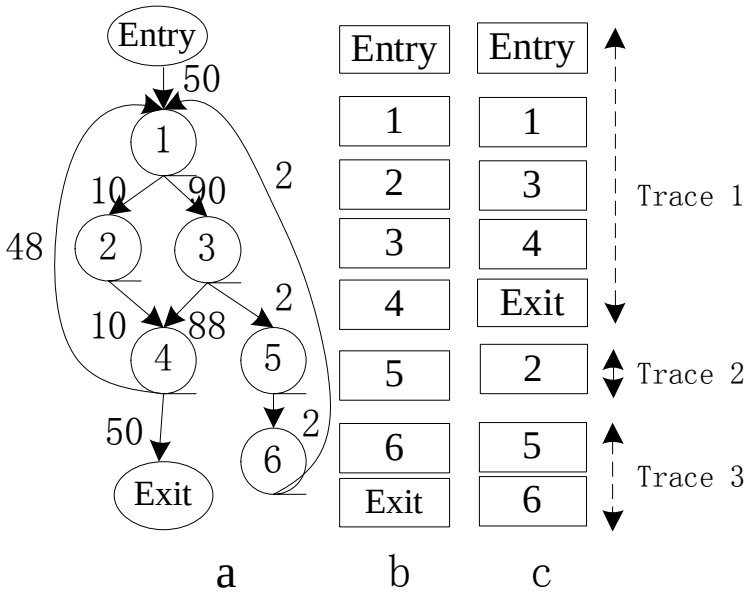

Figure 4.2: Intra-procedural layout optimization

4.3.2 Inter-procedural instruction layout optimization

Our inter-procedural layout optimization algorithm is shown in Algorithm 3. We extended the method proposed by Gloy et al [29] to perform inter-procedural instruction layout optimization. A procedure is considered to be a linear list containing several separate traces whose last basic block has only an out edge caused by an absolute jump instruction. We scan the traces sequentially, and add the scanned trace into the `dense_trace_list` until we encounter a trace whose first basic block has an execution count that is less than a factor of the execution count of the procedure. In our work, we empirically set the value of this factor as 0.05.

Most instruction layout optimization methods require a metric to keep track of the conflict between instruction blocks. In [75], the conflict metric is the *weighted call graph* (WCG), whose nodes represent the procedures, and whose edges specify the call relation between the procedures. The weights of the edges are the number of the function invocations. The WCG can only summarize direct call information, but it cannot capture the conflict between those procedures without call relations.

Algorithm 3: Inter-procedural instruction layout optimization.

Input: *Proc_list*: The list of all procedures of the application
Input: T: dynamic execution procedure call trace
Output: *Proc_list_opt*: a linear procedure list produced by inter-procedural instruction layout optimization

```
1  foreach p in Proc_list do
2  |  Get_size_of_dense_part(p);
   end
3  construct TRG using T;
4  Compute_popular_procedures;
5  foreach p in popular procedure list do
6  |  Split_procedure(p);
   end
7  Compute the cache-relative alignments for all dense parts of popular procedures;
8  Produce the final linear list: L;
9  return L%
10 Procedure Get_size_of_dense_part()
   Input: The procedure
   Output: A trace list(i.e. the dense part of procedure)
   Output: the size of dense part
11 Variable Num_Called : number of times procedure called;
12 Variable trace_list : list of all traces of procedure;
13 Variable dense_trace_list : list of all dense traces of procedure;
14 Variable trace_count : execution count of trace's first basic block;
15 foreach p in trace_list do
   |  if trace_count of p ≥ Num_Called * 0.05 then
16 |  |  add p into dense_trace_list;
   |  end
   end
17 Procedure Compute_popular_procedures()
   Input: TRG
   Output: the list of all popular procedures
18 Variable Total_Weights : sum of the weights of all the edges in TRG;
19 Variable list_edges : list of all edges in TRG, list_pop_edges : list of popular
   edges in TRG;
20 Sort list_edges by decreasing order of their weights;
21 for p from head of list_edges to tail of list_edges do weight += weight of p;
   if weight ≤ Num_Called*0.99 then
22 |  add p into list_pop_edges;
   else
   |  break;
   end
23 Output the nodes(i.e. procedures) of list_pop_edges;
24 Procedure Split_procedure(p)
   Input: p: procedure to be splitted
   Output: pd: dense part of procedure, and pl: loose part
25 get_size_of_dense_part(p);
26 pd = The dense trace list;
27 pl = p - The dense trace list;
```

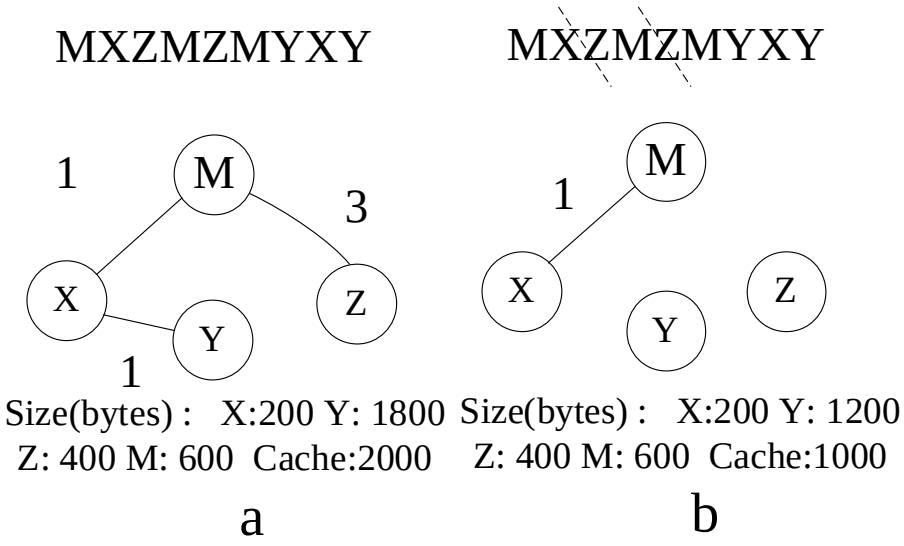

Figure 4.3: The temporal relationship graph.

An alternative conflict metric is the *temporal relationship graph* (TRG) [29]. The TRG is built dynamically and it can summarize the information on the interleaving of procedures in a program trace. The TRG is more general than the WCG in that the TRG can capture not only the conflicts between procedures with call relation, but also the conflicts between procedures without call relation. An example of the TRG is shown in Figure 4.3. Given a procedure call trace of an execution, the TRG can be built using a FIFO queue of a certain size, denoted as Q [29]. The procedure calls are taken one at a time from the procedure call trace and placed into Q. If there are two instances, denoted as p_1 and p_2, of a certain procedure call p in Q, then all the procedures between p_1 and p_2 are considered to be possibly in conflict with p. The weights of the edges between the procedures and p in the TRG is then incremented by one.

The size of all the procedures in Q should not exceed the size of Q, otherwise the earliest procedures will be expelled from Q. The reason is that if such a procedure were to be added to Q, all the cache entries have already been occupied by previous procedures and this procedure will definitely cause cache misses no matter how the

procedures are repositioned. The size of Q is proportional to the size of the cache. We empirically set Q to be 1.3 times the size of the cache. As in [29], we focused on optimizing the layout of the popular procedures.

Our algorithm differs from that of Gloy et al. in the following important ways. First, their algorithm repositions entire procedures during optimization. In our algorithm, we first split the popular procedures and then reposition only the denser parts of the popular procedures. The reason is that cache sizes in embedded system is generally small, thus it is necessary to compact the procedures to get better performance. Second, Gloy et al. used a chunk size of 256 instructions when computing the conflicts between two nodes of the TRG during the process of merging the nodes of TRG. However, we only consider the entire procedure. This greater granularity makes some parts of our algorithm simpler without significantly impacting performance.

4.4 Integrated instruction memory hierarchy design

The main drawback of traditional instruction layout optimization techniques is that they do not tune the memory hierarchy parameters according to the characteristics of the given application. For embedded systems, different applications might need specific memory hierarchy configurations to meet their requirements. Thus, it is possible to improve the effectiveness of instruction layout optimization by considering the instruction memory hierarchy exploration.

Similarly, it is not an ideal solution to perform instruction memory hierarchy exploration solely. In particular, partitioning the storage resource into a SPM and

an instruction cache has its own risks. After dividing the storage resource, the cache size becomes smaller. Thus, the instructions assigned into the cache may cause severe cache conflicts if they are not placed well. Effective instruction layout and placement to reduce cache conflicts is especially important.

4.4.1 Instruction memory hierarchy exploration

In our previous work, we proposed an algorithm for design space exploration of the instruction memory hierarchy in embedded systems [27]. We will summarize the key ideas of this algorithm here. For more details, please refer to the previous chapter.

Our algorithm makes use of the Loop-Procedure Hierarchical Graph (LPHG) to represent a program. The LPHG can capture the loops and the call relations between the procedures in the program. We developed a heuristic to divide the given storage resource into a SPM and a cache according to the program's characteristics. Then, we assign frequently executed instructions in the loops to the SPM, while attempting to minimize the conflict misses in the cache. The goals are to decrease the instruction cache conflict misses and to reduce the energy consumption of the overall memory hierarchy. However, the main drawback of our previous algorithm is that it did not consider instruction layout optimization in conjunction with instruction memory hierarchy exploration.

4.4.2 The combined approach

Now we will discuss how we combine the instruction layout optimization (ILO) and the memory architecture exploration (MAE) in our framework. Since these two techniques are inter-dependent, the phase order in which they are applied can affect

the overall performance. We consider two possible ways to apply these optimizations. In the first approach, which we named the MAE-first method, MAE will be performed first before ILO. On the other hand, in the second approach called the ILO-first method, ILO will be applied first and followed by MAE.

MAE-first method

Algorithm 4 shows the algorithm for the MAE-first method. The algorithm consists of several steps. In the first step, after obtaining the profiling statistics, we perform the intra-procedural instruction layout optimization using Algorithm 2. The benefits are as follows. First, it improves the instruction locality, and thus reduces the instruction cache miss. Second, it should decrease the cache conflict miss inside a loop because the frequently executed instructions within a loop will be placed together after the intra-procedural optimization. As a result, the loop is more likely to fit into the cache.

In the second step, we perform the design space exploration and instruction partitioning using the algorithm we proposed in [27]. This is because for the MAE-first method, we perform the MAE optimization first.

In the third step, after determining the configuration of the instruction memory hierarchy and partitioning the instruction to SPM, we recalculate the sizes of the partitioned procedures. This step is necessary to build the TRG used for inter-procedural optimization. After determining the memory architecture configuration and partitioning the instructions, the sizes of the procedures actually change. This is because for some procedures, their frequently executed instructions will be assigned into SPM. When we build the TRG, the total size of all the procedures in the Q will not exceed a factor of the cache size. The weights of the edges of the TRG will be affected by the sizes of the procedures.

Algorithm 4: Integrated memory hierarchy design : MAE-first method

Input: The given application: A, and the profiling statistics:$Prof$
Output: The optimized application and the instruction configuration
1 **foreach** *procedure(denoted as p) in A* **do**
  Perform intra-procedural optimization;
end
2 Design space exploration and instruction partitioning;
3 **foreach** *procedure(denoted as p) in A* **do**
  Get_size_of_dense_part(p);
end
4 Build TRG;
5 Recalc_weight_for_TRG;
6 Inter-procedural optimization using the TRG;7 Pack the instructions assigned to SPM;
8 Output the optimized application and the coupled instruction memory hierarchy.
9 **Procedure Recalc_weight_for_TRG**
Input: r: The list of original TRG edges
Output: s: The list of the updated weights of TRG edges
10 **foreach** *p in r* **do**
  foreach *nodes(q) of p* **do**
    if *the first trace of q assigned to SPM and the trace exec count* $<$ *q exec count* **then**
11 weight of p = weight of p * (trace exec count)/ q exec count;
    end
  end
end
End Procedure

If the size of a procedure is changed, it may occupy less cache blocks in the cache, which will tend to cause less cache conflicts with other procedures. We consider this factor when building the TRG. If the first separate trace is completely placed into SPM, we normalize the new partitioned procedure size as the size of the dense part of the procedure multiplied by a normalization factor, f. We define the normalization factor, f = min (original procedure execution count, execution count of first basic block of next remaining trace) / original procedure execution count. In the extreme case when the whole procedure is placed into SPM, there will not be any conflicts with other procedures. Such a procedure will not be put into Q during the process of building the TRG.

The fourth step is to build the TRG after partitioning the important instructions to SPM by considering the changes to the procedures' sizes. Figure 4.3 shows an example how to build the TRG after the instruction partitioning. The dynamic procedure execution trace is shown at the top of the figure, while the procedure size and cache size are shown at the bottom. Figure 4.3a is the TRG before instruction partitioning. Suppose that 2,000 bytes of storage resource is divided equally into SPM and cache, the whole procedure Z and 600 bytes of Y are placed into SPM. The procedure Z will not conflict with other procedures. Then, there is no edge between M and Z in the TRG of Figure 4.3b. Note that even if they are interleaved, there is no edge between X and Y in Figure 4.3b since the size of Y is larger than the Q size (set at 1.3 $\times$ cache size). The X and Y will always be in conflict no matter how we reposition them because Y (1,200bytes) is larger than the cache size (1,000bytes). Thus the weight between them should be 0, indicating the lowest importance during the layout optimization.

Once the TRG is obtained from the previous step, we need to recalculate the weights of the edges in the case that the first separate trace of the nodes of those edges have been partitioned into SPM. We change the weights of those edges by applying: $weight = f \times weight$. The reason is that after partitioning the frequently executed instructions of one procedure into SPM, it is less likely to cause cache conflicts with other procedures. So we decrease the weight to reflect this.

The final step is to perform the inter-procedural instruction layout optimization using the obtained TRG. This step is the same as described in Algorithm 3. The gap between the popular procedures are filled using non-popular procedures or the non-dense parts of the popular procedures. The output is a linear procedure list where all the procedures are properly placed and mapped into suitable cache addresses to minimize the cache conflicts.

ILO-first method

Unlike the MAE-first method, the ILO-first method first performs the instruction layout optimization and then does the memory architecture exploration and instruction partitioning. In Figure 1, this refers to the exchanging of the order of (a) and (b). Using the assembly code of the given application and the full given cache size, the instruction layout will be optimized by the method described in Section 4. After that, we will decide the instruction memory configuration and partition the instructions. To maintain the instruction layout during the instruction partitioning, we filled the memory locations for which the instructions have been moved to SPM with `nop` instructions.

4.5 Performance evaluation

4.5.1 Experimental methodology

We used the Simplescalar/PISA 3.0d simulation infrastructure [19] for our experiments. The most full-feature simulator in the suite, `sim-outorder`, was modified to support our instruction memory hierarchy consisting of SPM and cache. In our earlier work in which we used the Nios-II, we had used a cache line size of 32 bytes [27]. For this work, we used a direct mapped cache with cache line size of 64 bytes corresponding. This was necessary because the instruction size of Simplescalar/PISA is 8 bytes while that of the Nios-II is 4 bytes. Because 4 bytes of the 8 byte Simplescalar/PISA instruction is meant for user annotation and is therefore not part of the ISA, effectively all sizes of the SPM and caches reported below should be halved. The instruction hierarchy consists only of the L1 instruction cache (with or without the SPM) and main memory. Main memory is assumed pipelined. The latency

of the first access to the main memory is 10 cycles, while that of the subsequent accesses is 1 cycles. File I/O operations in the benchmarks were excluded from the execution statistics.

Our tool takes Simplescalar/PISA assembly files as input, reconstructs the CFG, instruments the code for profiling as well as performs layout optimization.

In our experiments, we used five application benchmarks from the MediaBench [61] and the MiBench [32] suites. We compared the performance and the energy consumption of executing each benchmark under three scenarios: (1) baseline: neither instruction layout optimization nor memory hierarchy exploration is performed; (2) ILO-only: the memory hierarchy only consists of instruction cache and the instruction layout of the application is optimized; (3) ILO-first: assuming that all the memory budget goes to the cache during the layout optimization phase before MAE; and (4) MAE-first. The missing scenario of 'MAE-only' has been evaluated in our earlier work [27].

We modeled energy consumption using the CACTI [99] model for $0.5\mu m$ technology. For the calculation of energy consumption, we included the logic elements connecting the instruction cache and the SPM to the processor. However, we excluded the part of the SPM controller that statically loads instructions into the SPM from the energy consumption calculation.

4.5.2 Performance improvements and energy savings

Table 4.1 shows the custom instruction memory hierarchy parameters for the various benchmarks as determined by our framework. "mem budgets" means the on-chip storage resource size for the instruction memory hierarchy. For each memory architecture configuration, the number with suffix 'C' stands for the instruction cache

Benchmk	mem budget(bytes)	custom arch
dijkstra	512	64C+448S
g721encode	4096	64C+4032S
g721decode	4096	64C+4032S
mpeg2decode	4096	2048C+2048S
pegwitencode	1024	512C+512S

Table 4.1: Application-specific memory configurations.

size, while the number with suffix 'S' represents the SPM size. All sizes are in bytes. We have deliberately chosen the budget so that it is insufficient to hold the entire working set, especially of the frequently executed parts, of the benchmarks.

Performance: The miss rate results for the three experiment setups are shown in Figure 4.4.

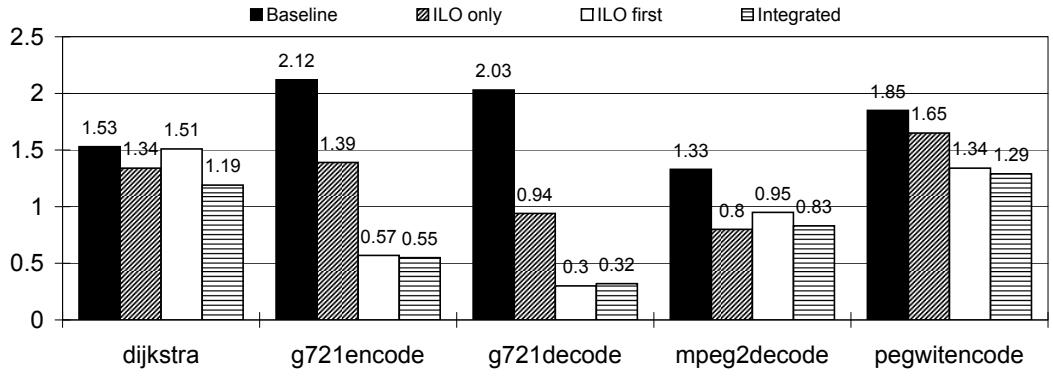

Figure 4.4: Instruction miss rates.

The miss rate for the third and fourth column is the instruction fetch misses in both cache and SPM divided by the total number of instruction fetches. As shown in the table, instruction layout optimization does quite well. Just by applying it, the miss rate was reduced by 10.8% to 53.7% with the average being 30.2%. The details are shown in Table 4.2. For ILO-first, the miss rate reduction ranges from 1.3% to 73.1%, with average value of 43.2%. These results show that combining the two optimizations bears significant fruits. By performing MAE-first, the average miss rate reduction is 49.7% with the actual miss rates ranging from 22.2% to 84.2%. This is better than the previous two methods.

These results demonstrate that combining the memory hierarchy exploration with instruction layout optimization can significantly reduce the instruction miss rate which in turn improves performance. The exception is `mpeg2decode` in which applying layout optimization alone produced lower miss rate than that of the integrated method. One possible reason is that instruction layout optimization has already effectively prevented a significant amount of instruction cache conflict misses. One effect of storage partitioning is that the remaining instructions not assigned to the SPM may experience a higher miss rate since the cache is reduced after partitioning. The heuristic algorithm tries to divide the storage resource when it estimates that the cache misses caused by dividing the resource is not increased too significantly. Since the SPM consumes less energy than cache, it may be profitable to relocate frequently instructions to SPM even if it means a slightly higher cache miss rate for the other instructions.

For `g721-decode`, ILO-first performed better than MAE-first. We investigated this further. We doubled the cache size by first doubling the number of sets, and next, keeping the number of sets the same as before, doubled the size of cache lines. For the former ILO-first continued to do better than MAE-first. However, for the latter, MAE-first did better. In fact, when we increased the cache line size to four times that of the original (i.e., 256 bytes), the miss rate for MAE-first was only 60% (87% in terms of overall execution cycles) that of ILO-first. We believe this is evident that MAE-first leads to a better spatial locality in the final code. However,

	miss rate reduction(%)			exe cycle reduction(%)			energy reduction(%)		
	ILO-only	ILO-first	MAE-first	ILO-only	ILO-first	MAE-first	ILO-only	ILO-first	MAE-first
dijkstra	12.4	1.30	22.2	4	-1	7	7	1	19
g721encode	34.4	73.1	74.1	21	42	43	30	66	67
g721decode	53.7	85.2	84.2	31	54	47	45	75	74
mpeg2decode	39.8	28.6	37.6	17	12	16	31	26	34
pegwitencode	10.8	27.6	30.3	5	10	10	10	25	27
Average imprv	30.2	43.2	49.7	15.6	23.4	24.6	24.6	38.6	44.2

Table 4.2: Miss rate, execution cycle, and energy consumption improvements.

for the work reported here, we did not consider the cache line size as a parameter for exploration.

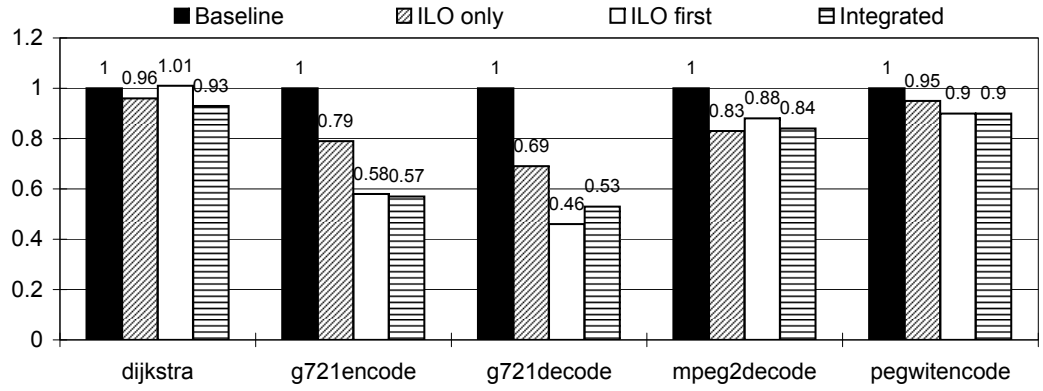

Figure 4.5: Normalized execution cycles.

As a result of the improvement in miss rate, the execution cycle counts of the applications are also decreased accordingly. As shown in Figure 4.5, the ILO-only method achieves an average of 15.6% reduction for the benchmarks studied, while ILO-first and MAE-first obtain an average of 23.4% and 24.6% respectively. Again the MAE method performs marginally better compared to the other methods.

Energy consumption: The normalized energy consumption results are shown in Figure 4.6.

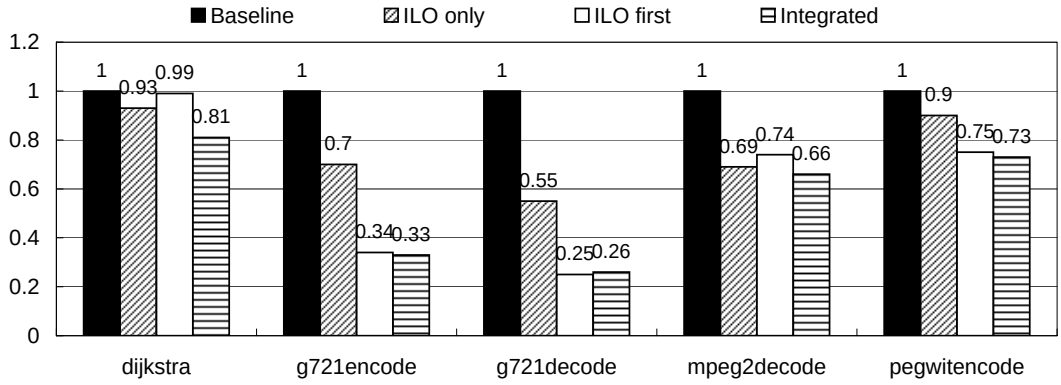

Figure 4.6: Normalized energy consumption.

From Figure 4.6, the ILO-only method can reduced the energy consumption for the benchmarks with an average reduction of 24.6% ranging from 7% to 45%

over the baseline. For ILO-first method, the average energy reduction is 38.6% with the actual savings ranging from 1% to 75%. The MAE-first method achieved an average energy reduction of 44.2%, with the actual savings ranging from 19% to 74%. The results shows that the MAE-first method yielded a better energy saving than the ILO-only method. Interestingly, for the `mpeg2decode` benchmark, the MAE method achieved more energy savings over the ILO-only method, despite a marginally worse off miss rate. The SPM consume less energy, thus relocating the frequently executed instructions into SPM more than offset the slight increase in the dynamic cache power caused by the higher miss rate.

4.6 Summary

There are two main methods to improve the instruction memory hierarchy performance. The first method is instruction layout optimization, and the second method is the architecture exploration to determine the optimal instruction memory hierarchy for a particular application. Previous work does not study the interaction between these two. We present SRIM that belongs to the latter in the previous chapter without considering combination of SRIM and instruction layout optimization.

In this chapter, we have developed a framework that combines instruction layout optimization and memory hierarchy exploration for embedded processors with two methods of integrating them, namely ILO-first and MAE-first. Given a specific application and a fixed amount of on-chip storage resource for the instruction memory hierarchy, our framework performs instruction layout optimization and instruction memory hierarchy exploration jointly to produce the global optimal result for maximizing the performance and minimizing overall energy consumption.

We believe ours is the first approach that considers both instruction layout optimization and memory hierarchy design space exploration in an integrated manner. Experimental results show that our proposed framework can achieve significant performance improvement and energy reduction over the traditional methods that failed to do so.

Chapter 5

DRIM: Dynamic Reconfigurable Instruction Memory

5.1 Motivation and related work

In previous chapters, we introduced a static reconfigurable instruction memory hierarchy design, SRIM, to improve performance and save energy. By applying SRIM, the given resource budget could be better used for an given application. The configuration of SRIM is determined according to the needs and characteristics of the application level. The static scheme is suitable those systems which run a single application and different phases of the execution of the application require similar memory configurations. If there are multiple applications and multiple phases with various features within an application, the static scheme will become inflexible to meet the requirements for different memory configurations from the applications and phases.

In order to meet the above requirements, we propose a novel dynamic reconfigurable instruction memory hierarchy (DRIM) [28] for embedded systems. Our

proposed architecture consists of four banks of storage, each of which can be dynamically reconfigured to be part of a cache or a SPM to suit an application and its execution phases.

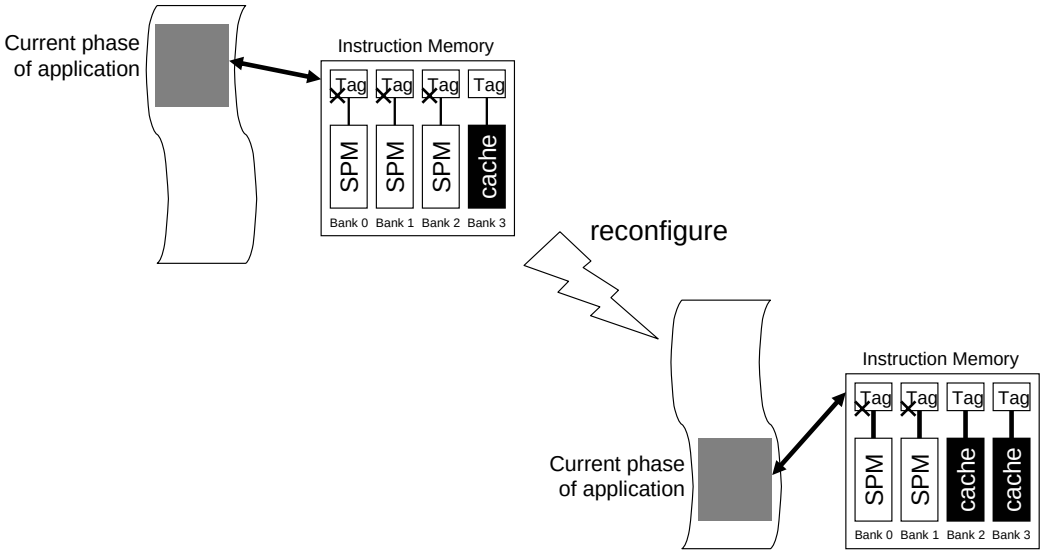

Figure 5.1: Reconfiguring memory at runtime.

Figure 5.1 illustrates the idea of how the DRIM architecture works. Differences between applications as well as between phases of execution within an application can best be exploited if the memory hierarchy can be reconfigured. In the configuration of DRIM that we designed and studied, we reconfigured four banks of storage dynamically as cache or SPM according to the needs of different applications and phases.

To support DRIM, we will also describe an algorithm that supports the dynamic reconfiguration of DRIM and the selection and allocation of code to be executed from the scratchpad memory. Our experimental results using six benchmarks from the Mediabench and the MiBench suites show that our framework can achieve significant energy savings.

Related work Several approaches have been proposed for the reduction of energy consumption in caches. First, there were proposals for customizable and reconfigurable caches that adapt to the characteristics of a specific application. Cache banks are shut down and the cache associativity is reconfigured when necessary in order to decrease energy consumption [103, 6].

Some other researchers have studied the use of SPM in the instruction memory hierarchy, with the aim of saving energy in embedded systems. Instruction may be statically mapped into a given instruction memory hierarchy consisting of only of SPM or a mixture of cache and SPM [97, 8]. Algorithms to statically partition instructions for a SPM of a given size (with or without the presence of a cache) have been proposed. However, these works do not consider dynamic instruction replacement and the possibility of changing hardware configurations. Such schemes suffer from a lack of flexibility and the SPM is not efficiently used.

Dynamic instruction replacement to improve the utilization of SPM has also been studied [93, 47, 81]. Different instruction blocks may occupy and reuse the same SPM entries to improve the SPM's efficiency. Significantly greater energy savings can be achieved over the static methods. However, the above works did not consider tuning the architecture parameters for different applications. Several other researchers studied the problem of design space exploration so as to find the best memory hierarchy parameters for a given application. Vander et al. [5] performed the exploration for optimal configurations of the instruction loop buffer given an application, and mapped selected loops into these loop buffers so as to reduce the energy consumption.

The existing work on instruction SPM can be classified into three categories: (i) static architecture with static mapping, (ii) static architecture with some dynamic replacement strategies, and (iii) static architecture exploration with static mapping.

None of these considered the dynamic tuning architectural parameters. Kondo et al [60] proposed a dynamic reconfigurable data memory hierarchy consisting of SPM and cache. However, they did not consider the instruction memory hierarchy. The main contributions of our work are as follows:

1. We propose a new dynamic reconfigurable instruction memory hierarchy (DRIM), which enables more flexible use of a SPM than previous methods. It can be reconfigured for different applications instead of being tuned just for a particular program. Furthermore, DRIM can be reconfigured for the different phases of execution of an application, so as to minimize the energy consumption of each phase.

2. We developed a compilation strategy to support this reconfiguration memory hierarchy.

To the best of our knowledge, dynamic reconfiguration of SPM for instruction memory hierarchy has yet to be studied.

5.2 DRIM architecture

Differences between applications as well as between phases of execution within an application can best be exploited if the memory hierarchy can be reconfigured. Figure 5.1 illustrates the idea of how the DRIM architecture works. In the configuration of DRIM that we designed and studied, we reconfigured four banks of storage dynamically as cache or SPM.

The architecture of DRIM is shown in Figure 5.2. It consists of the tag logic, the data array, the SPM control logic, and other logic. The DRIM architecture is

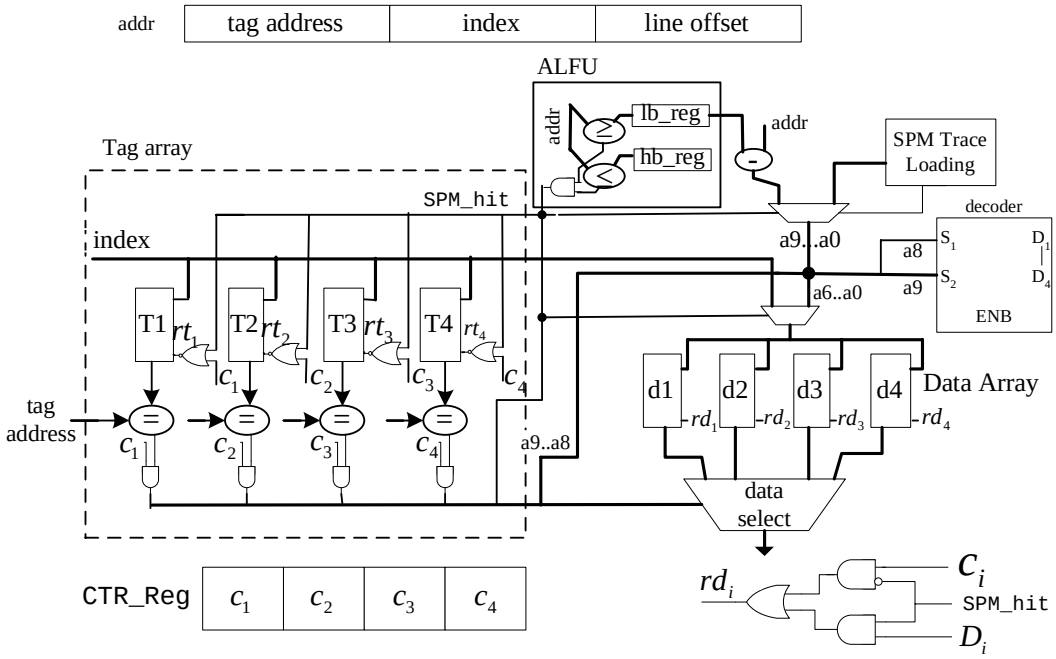

Figure 5.2: DRIM architecture.

based on a four way associative cache architecture. One important difference is that tag and data access is controlled by the SPM control logic and a control register known as `CTR_Reg`. In Figure 5.2, the `CTR_Reg` is collectively the four bits, c_1, to c_4. These four bits determine the configuration of DRIM. Each bit is associated with one bank of tag and data storage. If the bit is one, the corresponding data bank will be configured as a SPM. Also, the tag bank will be gated, thereby decreasing its activity, which in turns results in energy savings. The value of the `CTR_Reg` is manipulated by the processor.

The *address lookup functional unit* (ALFU) determines whether an instruction is residing in the SPM or not. It consists of two address registers and two parallel comparators. The two registers, `ub_reg` and `lb_reg`, hold the upperbound and the lowerbound addresses for the instruction block that is to reside in the SPM, respectively. If the address of an instruction to be fetched falls within the range of these two registers, then it is in the SPM and the ALFU will generate the `SPM_hit` signal. This signal controls the selection and gating of the tag and the data banks.

In the DRIM design presented here, there is only one pair of `ub_reg` and `lb_reg`. As a result, only one block of instructions may reside in the SPM banks at any one time.

The SPM controller performs the loading of instructions from main memory into the SPM, as well as the updating of the upperbound and lowerbound registers.

The tag and data banks are selected or gated according to the value of `CTR_Reg`. The address of the instructions in SPM will determine which SPM bank should be accessed. For each tag array t_i, the gate signal is rt_i:

$$rt_i = \sim\texttt{SPM\_hit} \wedge \sim c_i = \sim (\texttt{SPM\_hit} \vee c_i)$$

In other words, all the tag banks will be gated and de-activated if `SPM_hit` is asserted. A de-asserted `SPM_hit` implies that the instruction to be fetched is not in the SPM. In that case, only the tags corresponding to the banks configured as cache, i.e., those whose c_i is true, will be searched.

For data array i, the corresponding gate signal, rd_i is:

$$rd_i = (\sim\texttt{SPM\_hit} \wedge \sim c_i) \vee (\texttt{SPM\_hit} \wedge D_i)$$

where D_i is the data bank selection signal. If an instruction is not in SPM, i.e. `SPM_hit` is false, the data array of the storage banks configured as cache will be accessed. Otherwise, i.e. if `SPM_hit` is true, the SPM bank containing the instruction will be selected by D_i. The following simple example illustrates how D_i can be computed: Suppose all four data banks are configured as SPM and the size of each data bank is 256 bytes. In this case, bank 1, 2, 3, and 4 will hold instructions for which the last 10 bits of the addresses are in the range 0x000 to 0x0FF, 0x100 to

0x1FF, 0x200 to 0x2FF, and 0x300 to 0x3FF, respectively. Clearly, the two most significant bits can then be used as the bank selection signal D_i. The remaining eight bits can be used as the address supplied to the data banks.

5.3 Compiler framework

DRIM requires the compiler's support to realise its dynamic reconfiguration. The compiler also has to insert instructions into an application in order to dynamically load selected instructions into the SPM. We have developed a compiler framework that performs these functions.

5.3.1 Compilation flow

The structure of our compilation flow is shown in Figure 5.3. The inputs are the given application and the storage resource budget for the instruction memory hierarchy. The outputs are the partitioning decision for the instruction memory hierarchy custom-made for the application, and the transformed application with an optimized instruction layout. The framework consists of several steps:

- *Profiling the application*: First, profiling is used to obtain the runtime characteristics of the application. The information collected include the execution counts of the edges of the control flow graphs (CFGs) of all the procedures and the number of the procedure invocations. This is done by building a CFG for each procedure, and then adding instructions to instrument each basic block of a CFG. The instrumented program is executed to get the required execution statistics.

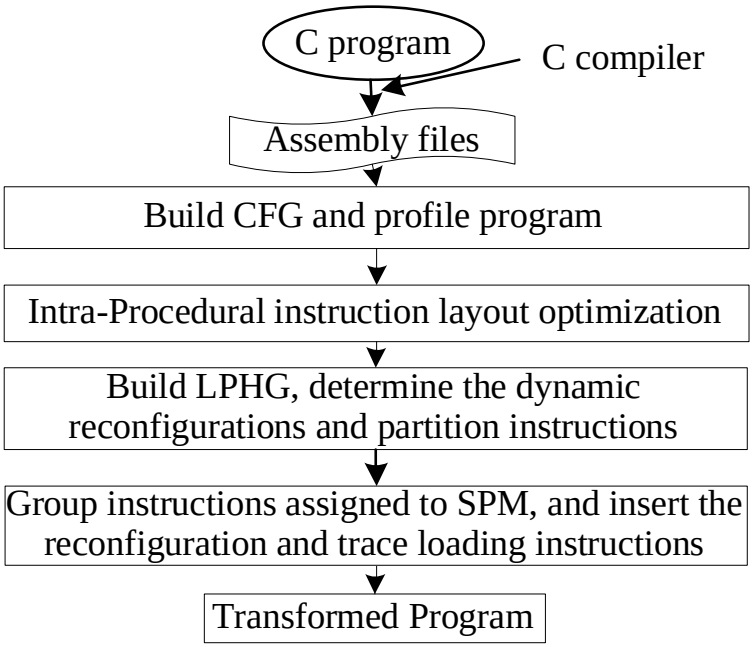

Figure 5.3: Design flow

- *Intra-procedural instruction layout optimization*: The goal of this step is to optimize the instruction layout within each procedure according to the profiling statistics obtained in the previous step. We used the Top-down Positioning algorithm proposed by Pettis and Hansen [75] to perform intra-procedural layout optimization. This step brings the frequently executed basic blocks together to make it easier to extract a frequently executed trace.

- *Determining the reconfiguration and partitioning instructions to SPM*: In this step, the application's runtime profile is analyzed so as to determine the suitable points to reconfigure DRIM and the corresponding configurations. At the same time, the instructions blocks are partitioned to the dynamically configured SPM banks.

- *Grouping partitioned instruction blocks, and inserting reconfiguration and trace load instructions*: After the preceding step, the architectural configurations for different phases are determined and the instructions are partitioned to SPM banks. At this step, we generate code chunks namely traces by taking out and

grouping the instruction blocks assigned to SPM. Then, the instructions for architecture reconfiguration and trace loading are inserted into the application. All the instructions in an trace are contiguous and the whole trace will be loaded into SPM in case a loading happens. The jump instructions might need to be added to maintain the control flow relations between basic blocks.

We evaluate the proposed framework using the Simplescalar tool set [19]. The Simplescalar simulator was extended to support DRIM. We also built an instruction optimization tool which performs the program profiling and the intra-procedural instruction layout optimization.

5.3.2 Dynamic reconfigurations and instruction replacement

This section describes the second innovation of this work other than the DRIM architecture, namely an algorithm to decide where and when to reconfigure DRIM as well as deciding which instructions should go into the SPM. The reconfiguration and the instruction allocation are determined by the phaseal behavior of the execution of an application. Our proposed algorithm is shown in Algorithm 5. The algorithm uses the Loop-Procedure Hierarchy Graph (LPHG) [64] to represent a program. The LPHG captures all the loops, and procedure calls of an application as well as their relations. In order to estimate the cache misses for loops, the sizes of loops in LPHG are computed (line 2 of Algorithm 5).

Algorithm 5: Algorithm for determining dynamic reconfiguration and SPM instruction load points

Input: *Proc_list*: Procedure list whose procedures have been intra-procedural optimized
Output: *Basic_block_list*: list of instruction blocks of basic blocks assigned to SPM
Variable *list_loops* : the list of loops;
Variable *list_child_loops* : the list of loops;
Build Loop-Procedure Hierarchy Graph(LPHG);
Get_sizes_of_all_loops();
list_loops ⟵ all leaf loops;
foreach *loop l in list_loops* **do**
    if *((l is leaf loop) && (#iteration of l ≥ Thresh_hold))* **then**
        Annotate_reconfig_point_and_instrs_partitioned(l);
    else if *l is non-leaf loop* **then**
        list_child_loops ⟵ all child loops of l;
        #banks_occcupied = # of banks configured as SPM for loops in *list_child_loops*;
        $\#free\_banks = \#total\_banks$ - *#banks_occcupied*;
        $\#SPM\_banks$ = evaluateConflict($\#free\_banks$, $child\_loops\_of\_l \cup l$);
        if *(#SPM_banks !=0)* **then**
            Instr_alloc($list\_child\_loops \cup l$, SPM);
            Update_reconfig_point(l);
        end
    end
    if *(!list_loops.contain(parent_of(l)))* **then**
        list_loops.add_to_tail(parent_of(l)); //traverse parent loops in the future
    end
end
Hoist_reconfig_position();
Insert_reconfig_and_code_loading_instructions();
return *Proc_list*;

We assume that most of the energy consumed by instruction fetching as well as most of the instruction cache conflicts occurs inside loops. The intuition is that if the number of loop iteration are large enough to outweigh the overhead of the reconfiguration and trace loading, then the loop should be placed into the SPM. If the loop is too big to fit into the SPM, then the cache is used to buffer the rest of it.

In a LPHG, the deeper a loop is, the higher is its execution frequency. The algorithm therefore starts from the leaf loops and work toward their parent loops. If the number of the loop iterations is larger than a threshold value, the energy savings obtained from the usage of SPM will overweigh the overhead of reconfiguring DRIM. It is then beneficial to reconfigure the data storage banks into SPM and use it. For this work, we empirically set the threshold value to be 30.

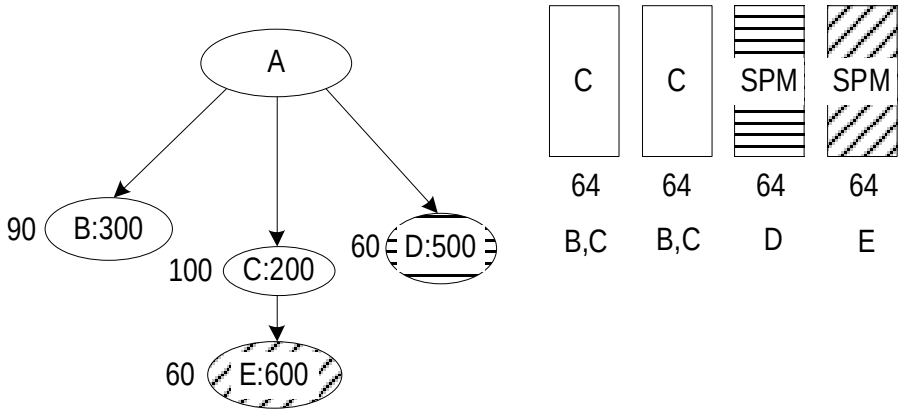

Figure 5.4: Example of loop allocation.

After a loop is examined, its parent loop will be added to *list_loops* (line 14). The algorithm may at some later point examine it for more opportunities for reconfiguration. The algorithm therefore proceed one level at a time from leaves up to the root. If a loop is an internal node (line 5), then the algorithm will evaluate whether it is beneficial to allocate more SPM space from the free storage banks (line 9). The evaluation function we used is conservative and simple. If the reduction in cache size caused by the allocation of more space to SPM does not severely increase the instruction miss rate, then it is considered beneficial. The evaluation function takes the number of free storage banks for reconfiguration and the current loop as input. It returns the maximal number of additional SPM banks ($\#SPM\_banks$) which can yield beneficial results.

Figure 5.4 is a example of how the algorithm evaluates conflicts and partitions the instructions. The left part of Figure 5.4 is a sample loop represented in LPHG,

while the right are the four banks storage resource available. The algorithm first try to configure one bank as SPM and allocate it to loop E. Each of the left three child loops (i.e. B, C, D) can fit into the remaining three storage banks, i.e. there will not be any conflict. So, the algorithm will try to configure one more banks as SPM and move loop D, the loop with the next highest execution frequency, to it. Now, B and C, taken together, is smaller than the size of the two storage banks, and thus it is safe to take this configuration. If one more bank is configured as SPM, then there will only be one bank left to buffer the remaining loop and other code. It is therefore not beneficial to configure banks as SPM any more since severe cache conflicts will be caused with one of the loops.

The instruction allocation function allocates the frequently executed instructions inside the loop to the allocated SPM (line 11). The instruction allocation function considers two factors. The first is the size of the loop. If it is larger than the size of the allocated SPM, then as many instructions as possible of the loop will be allocated to the SPM. The second consideration is the execution frequency of instructions. The most frequently executed instructions will be allocated to the SPM.

After instruction allocation, all reconfiguration points inserted in the child loops by the previous iteration will be deleted and a new reconfiguration point is added to the entry of the loop (line 12). This is because before a `SPM_load` instruction loads a block of code, the child loop should not load another instruction block. There can only be one block of instructions residing in the SPM. The instructions loaded to SPM are frequently executed. Therefore care must be taken to avoid overlapping loops in the SPM.

Once all the loops are traversed and the reconfiguration positions and instructions assigned to SPM have been decided, the instructions for reconfiguration and

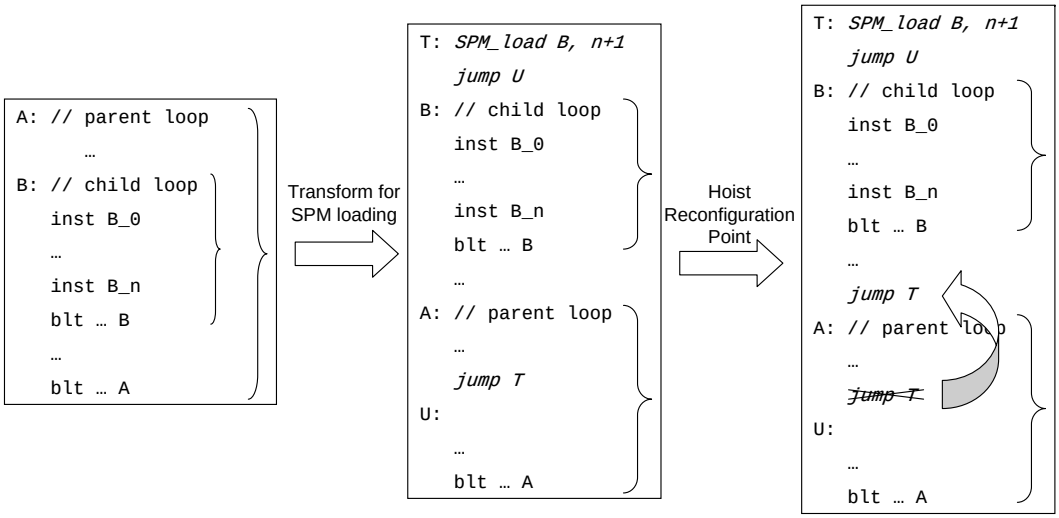

Figure 5.5: Code transformation for reconfiguration.

instruction loading are inserted. There is an important optimization that can be applied. The number of reconfigurations can be reduced by hoisting the reconfiguration point from inner loop to outer loop (line 15). If a loop does not have any sibling loops, the reconfiguration at its entry can be hoisted out to its parent loop. An example of this code transformation is shown in Figure 5.5. The `SPM_load` instruction loads a block of code into the SPM as well as set up the bound registers.

The last step in the algorithm (line 16), is to group all the instructions allocated to the SPM for each reconfiguration and insert the instructions used for reconfiguring the DRIM as well as loading the instruction blocks to SPM after each reconfiguration yielding the final transformed program.

5.4 Experimental evaluation

5.4.1 Experimental methodology

We used the Simplescalar/PISA 3.0d simulator [19] for our experiments. The full-featured simulator in the suite, `sim-outorder`, was modified to support DRIM. The

cache line modeled in the simulator is 64 bytes, corresponding to 32 bytes in 4-byte instruction systems. The instruction memory hierarchy consists only of the L1 instruction buffer (i.e. DRIM) and the main memory. Our DRIM implementation has four banks of data storage, each of size 256 bytes. The latency of accessing DRIM is 1 cycle. The main memory is assumed to be pipelined. The latency of the first access to the main memory is 10 cycles, while that of the subsequent accesses is 2 cycles.

In our experiments, we used six application benchmarks from the MediaBench [61] and MiBench [32] suites. We compared the energy consumption and performance of executing each benchmark on two different architectures: (1) a baseline system comprising of a traditional 4-way associative instruction cache and (2) a DRIM based system.

We modeled the energy consumption of the memory hierarchy using the CACTI [99] model for $0.13\mu m$ technology. For the calculation of the energy consumption of DRIM, we included the logic elements that perform address checking and control the SPM. The energy consumption of loading a trace into the SPM is modeled as the number of SDRAM burst accesses up to the size of the trace. The dynamic energy consumption per access of different architectures is shown in Table 5.1. '1way', '2way', '3way' and '4way' represents the energy consumption of the cache portion when DRIM is configured as a combination of 1, 2, 3, or 4 banks cache and the SPM respectively. 'SPM' is the per access energy consumption for the SPM in DRIM. This is the sum of the energy consumption for one data bank of the 4-way associative cache and the energy overhead for accessing the SPM. The energy consumed by each burst access of SDRAM is 32.5 nJ [90].

	DRIM					
base cache	1way	2way	3way	4way	SPM	SDRAM
0.538	0.152	0.283	0.413	0.544	0.133	32.5

Table 5.1: Per access energy consumption (in nJ).

5.4.2 Performance improvements and energy savings

Performance: The performance results are shown in Table 5.2. Compared to the baseline cache configuration, the decrease in the instruction cache miss rate provided by DRIM ranges from 0% to 40.7% for the benchmarks studied. The average improvement in the miss rate is 15.6%. This improvement comes from reconfiguring some storage banks to SPM and the mapping of the frequently executed instructions into the SPM for important loops. For the benchmarks `mpeg2-dec` and `mpeg2-enc`, there is no improvement on the miss rate because they are dominated by small size loops with very large number of iterations. Such benchmarks performs well on a pure cache architecture. As a result of the improvement in miss rates, the execution times of the applications are decreased by an average of 10.2%.

	miss rate(%)			execution cycles(K)		
Benchmark	baseline	DRIM	Imprv	baseline	DRIM	imprv(%)
gsm-dec	0.42	0.40	4.8	7,617	7,603	0.2
gsm-enc	6.10	3.62	40.7	70,076	47,633	32.0
g721-enc	3.09	2.43	21.4	381,509	331,266	13.2
susan-edge	2.76	2.03	26.4	2,346	1,962	16.4
mpeg2-dec	1.36	1.36	0.0	27,329	27,427	-0.4
mpeg2-enc	0.11	0.11	0.0	836,006	836,121	-0.0
average	-	-	15.6	-	-	10.2

Table 5.2: Miss rate and performance

Energy consumption: The total energy consumption of the two instruction memory hierarchies are shown in Table 5.3. Compared to the baseline cache cofiguration, the reduction in the energy consumption provided by DRIM ranges from 14.3% to 65.2% for the benchmarks studied. The average reduction in the energy consumption is 41%.

	gsm-dec	gsm-enc	g721-enc	susan-edge	mpeg2-dec	mpeg2-enc
baseline(mJ)	8.39	98.34	558.52	3.27	37.39	1,019.7
DRIM(mJ)	4.60	53.84	336.3	2.08	32.04	354.75
improv(%)	45.2	45.2	39.8	36.5	14.3	65.2

Table 5.3: Energy consumption.

There are two major reasons for the reduction in energy consumption. First, the instruction cache miss rate has improved. The per access energy consumption of SDRAM is much higher than that of the cache and SPM. Thus, fewer cache misses will translate to energy savings. Second, the per access energy consumption of the SPM is lower than that of the cache. By configuring one or more instruction storage buffer as SPM and loading the frequently executed instructions into them during the program execution, significant energy savings can be obtained. For example, although there were no miss rate reduction for `mpeg2-dec` and `mpeg2-enc` (as shown in Table 5.2), there is actually energy savings. `mpeg2-enc` has a higher energy reduction than `mpeg2-dec` since its miss rate is very low and the energy consumption is dominated by on-chip instruction buffer accesses. By reconfiguring on-chip storage buffer banks as SPM, the total energy consumption is decreased significantly.

5.5 Summary

In previous chapters, we introduced static reconfigurable instruction memory hierarchy to customize the architectural parameters for specific applications in the aim of achieving performance and energy improvement. The drawback of the static method is that the architectural parameters cannot be changed during runtime for different phases of an execution. In this chapter, we presented a low power dynamically reconfigurable instruction memory hierarchy, called DRIM, for embedded systems. The on-chip instruction storage banks can be reconfigured as SPM or cache for different applications as well as different phases of the application's execution.

We also developed a compilation flow to support DRIM. Our experimental results showed significant energy savings as well as satisfactory performance improvement. We believe that our approach is more flexible than previous schemes.

Chapter 6

PRIM: DVS-based Pipelined Reconfigurable Instruction Memory

6.1 Motivation and related work

In the previous chapter, we present a dynamic reconfigurable instruction memory hierarchy namely DRIM for energy reduction. The storage banks of DRIM can be reconfigured as SPM in order to decrease energy consumption. Apart from DRIM, several other reconfigurable memory architectures have been proposed to reduce the energy consumption of caches in embedded processors. For example, application-specific cache customizations allow the cache configuration parameters, such as associativity, line size, cache size, to be adapted to specific applications and/or their execution phases [6, 103]. Among them, shutting down idle cache banks is one of the most popular methods to reduce the energy consumption. However, the capacity of the idle cache banks is wasted. Is there any way to make use of this idle storage resource for further energy savings over the resource disabling? Our

answer is affirmative. We observe that operating an idle cache bank I at reduced voltage/frequency level along with an active bank A can potentially achieve better energy savings compared to shutting down I and operating A in normal mode. The key to maintain the performance is to pipeline and synchronize the accesses to these two banks through the appropriate instruction memory layout. Moreover, static analysis should determine appropriate program phases where one can achieve significant energy savings with minimal performance overhead by switching to low power mode. Towards this, we propose a novel DVS-based pipelined reconfigurable instruction memory hierarchy called PRIM to take advantage of unused resource to save more energy.

Several approaches have been proposed to reduce the energy consumption of caches in embedded processors. For example, application-specific cache customizations allow the cache configuration parameters, such as associativity, line size, cache size, to be adapted to specific applications and/or their execution phases [6, 103, 28]. Dynamic voltage scaling (DVS) [23] is another effective technique for power reduction where the supply voltage and clock frequency are dynamically adjusted in accordance to the need of the application. DVS leads to significant energy savings as the dynamic power (which dominates total power) of CMOS circuits is proportional to the square of the supply voltage and varies linearly with clock frequency.

Researchers have shown that most applications have multiple phases with varying resource requirements [86]. Resources that are under-utilized at a certain phase of an application can be exploited to further save energy consumption. The different approaches to save energy through exploitation of under-utilized resources can be broadly classified into two categories. The first class of approaches are *reactive* in nature in that they use DVS to slow down or completely shut down a resource when it becomes under-utilized. The second category of approaches *predict* the resource utilization and employ DVS to exploit the idle capacity of the under-utilized resources

in order to save energy while maintaining performance. Proposals for each of these two methods have been made for computational units [63, 11, 40]. For example, Chandrakasan and Brodersen [23] proposed an aggressive technique that replicates a logic block number of times with each instance operating at a lower supply voltage and frequency. In the context of memory hierarchies, most of the previously proposed techniques belong to first category of reactive approaches. Examples include selective cache ways [6], drowsy cache [56] and cache decay [51] mechanism that either disable the functioning of under-utilized memory resource to save energy.

This chapter presents for the first time a predictive DVS-based energy savings technique in memory hierarchy design. We propose a novel *pipelined reconfigurable instruction memory hierarchy* (PRIM) for power constrained embedded processors. Our canonical example of PRIM consists of an instruction cache with four banks for data storage, where two banks can be dynamically reconfigured to DVS mode. In DVS mode, the supply voltage and frequency of the two banks are deceased and the two banks become a low power buffer pair. Moreover, the low-voltage banks are operated as scratchpad memory that are explicitly controlled through software. In particular, frequently executed instructions are loaded and locked into the two DVS banks. Finally, instruction throughput is maintained by pipelining and alternating the instruction fetches between these two banks. We also developed a profile-driven algorithm that can analyze the application and insert appropriate cache reconfiguration points to support PRIM architecture. Our experimental results confirm that PRIM can achieve significant energy reduction for popular embedded benchmarks with minimal performance overhead. To the best of our knowledge, this is the first work that proposes taking advantage of under-utilized storage resources to obtain more energy savings instead of disabling them.

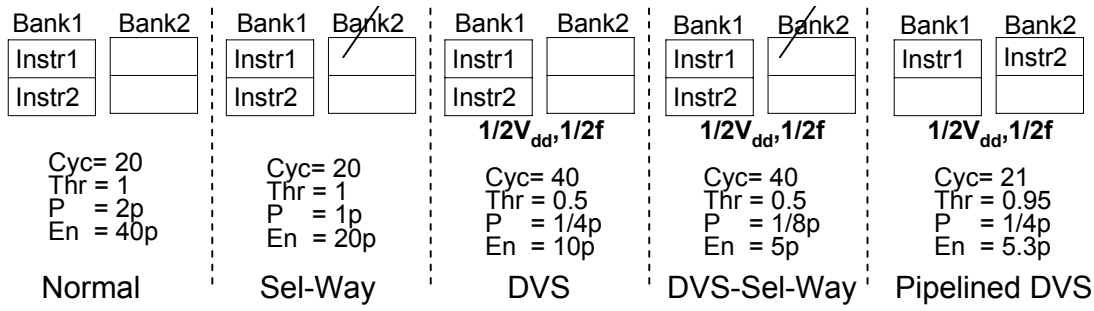

Figure 6.1: Comparison of cache energy savings techniques.

Motivating Example Let us illustrate the intuition behind PRIM and its advantage over existing cache energy savings techniques with a motivating example. Consider the execution of a small loop containing only two instructions from a 2-way set associative cache for 10 iterations. For simplicity of exposition, let us assume that each cache block contains only one instruction. Finally, both the instructions are assumed to be present in the same cache bank (way/column). Figure 6.1 shows a comparison among the different schemes based on (a) total cycles for instruction fetch (*Cyc*), (b) instruction fetch throughput (*Thr*), (c) power (*P*) and (d) total energy consumption (*En*) for one complete execution of the loop. We assume 1-cycle cache latency in normal mode and 2-cycle cache latency in DVS mode (as DVS mode runs at half the frequency). The dynamic power of each normal cache bank is $p = CV^2f$.

In normal mode, it will take 20 cycles to fetch 20 instructions with a throughput of 1 instruction/cycle. As the normal cache has two banks, power $P = 2p$ and energy $En = 2p \times 20$. In the *selective-way cache*, Bank2 is switched off as all the accesses go to Bank1. So th selective-way scheme can achieve 50% power and energy reduction compared to normal cache without sacrificing performance.

A more aggressive technique applies *DVS* to both the cache banks. The power requirement of DVS scheme is $P = 2 \times C \times (\frac{V}{2})^2 \times \frac{f}{2} = \frac{p}{4}$. However, as frequency is halved, the instruction fetch latency increases to 2 cycles and throughput drops

to 50% of the normal cache. The overall energy consumption of DVS scheme is $40 \times (\frac{p}{4})$, which is 50% savings compared to selective-way cache. To save additional power, the unused bank can be switched off in DVS mode as well leading to *DVS selective way* cache. Clearly, DVS selective way has the same performance as DVS scheme but reduces energy consumption further by 50%.

The main drawback of DVS or DVS selective way caching is that instruction throughput is affected adversely. This leads to poor IPC (instructions per cycle) in the processor and hence worse performance. Our key observation is that in DVS mode, one cache bank is not utilized and it either sits idle or gets switched off. Instead, we exploit this under utilized bank to maintain instruction throughput comparable to the normal cache in PRIM. Instead of powering down one cache bank for this loop, PRIM (a) distributes the instructions to the two cache banks and (b) powers up both the cache banks in DVS mode. We then propose the use of pipelined instruction fetches to hide the latency introduced by DVS thereby achieving the one instruction fetched per cycle throughput of the normal cache. An example of this pipelined access is shown below.

Cycle :	$C1$	$C2$	$C3$	$C4$	$C5$	$C6$	...
Bank1 :	$i1$		$i1$		$i1$	...	
Bank2 :		$i2$		$i2$		$i2$	...

Clearly, the throughput of 1 instruction per cycle is sustained. However, an accurate next instruction address prediction scheme needs to be in place in order for this to succeed.

In summary, PRIM has the same power consumption as the DVS scheme. However, the careful overlapping of accesses to the two banks reduces the total cycles requirement to 21 (1 additional cycle is required for pipeline initialization). Thus

the total energy consumption for this scheme is $En = \frac{p}{4} \times 21 = 5.3p$, which is similar to DVS selective way scheme (the best scheme in terms of energy) while the performance of PRIM is similar to that of the normal cache. Moreover, with larger loop sizes and greater number of iterations, the pipeline initialization overhead will become negligible.

Related work A lot of research have been done on the memory hierarchy with the aim of saving energy. Schemes were proposed to shut down certain number of cache ways when the cache is under-utilized [6, 78]. Zhang [103] proposed a highly reconfigurable cache whose associativity can be dynamically changed.

Many schemes and algorithms utilizing dynamic voltage and frequency scaling have also been developed for low energy processors. In [63, 11], the worst-case slack time and workload-variation slack time were exploited using DVS. When the utilization computed based on the WCETs of tasks is lower than 1, it may be possible to run some of tasks at a slower clock and a lower supply voltage without delaying their deadlines. Some researchers [11, 39] developed compilation flows that detect code regions for DVS and determine the optimal frequency scaling. There are also many proposals to apply DVS to the memory hierarchy. Kim et al [56] proposed the drowsy cache architecture where cache lines will be put into drowsy mode if they are not accessed for a certain time interval. This technique is quite similar to the selective way cache [6] in which under-utilized cache lines are set to a low power mode. The main difference between them is that the drowsy cache can hold data when switching between low and normal power modes. All these previous works on memory hierarchy management tried to disable under-utilized memory resource functioning to prevent these parts from consuming energy. Instead of disabling under-utilized resource, PRIM tries to make use of these resource in DVS mode so as to achieve retain the energy savings while recovering the performance loss.

Organization of the chapter We will now proceed to present the PRIM architecture. The motivating example illustrates that PRIM needs co-operation from the compiler in statically identifying program phases with under-utilized cache banks. Once these program fragments have been identified, the compiler inserts appropriate instructions to load and lock the content to the cache banks and switch the banks to DVS mode. At runtime, the execution simply follows these cache configuration hints to switch the cache between normal and DVS modes. Section 6.3 presents this compilation framework. We present our experimental methodology and discuss the results in Section 6.4. Our experimental results using six benchmarks from the Mediabench and the MiBench suites show that PRIM can achieve significant energy savings without degrading the performance. Finally we summarize the chapter (Section 6.5).

6.2 The PRIM architecture

Architectural considerations: We present the PRIM architecture based on a 4-way set associative cache as shown in Figure 6.2. PRIM architecture can achieve the following functionalities: (1) able to run DVS mode and normal cache mode according to execution needs (2) dynamically switch running mode (3) reload instruction blocks to DVS banks at runtime (4) the instruction fetches to DVS banks are pipelined in order to maintain throughput. We currently only emphasize on the situation of single application execution that cannot be interrupted by other applications and the application controls PRIM throughout its execution. The scenario of context switching for multiple threads is left for future study.

Control of DVS mode and normal cache mode: The cache consists of four data banks, the DVS control logic, the voltage-frequency controller (Vdd-Frq Ctrl), the address lookup function unit (ALFU), and the next-PC predictor. Unlike

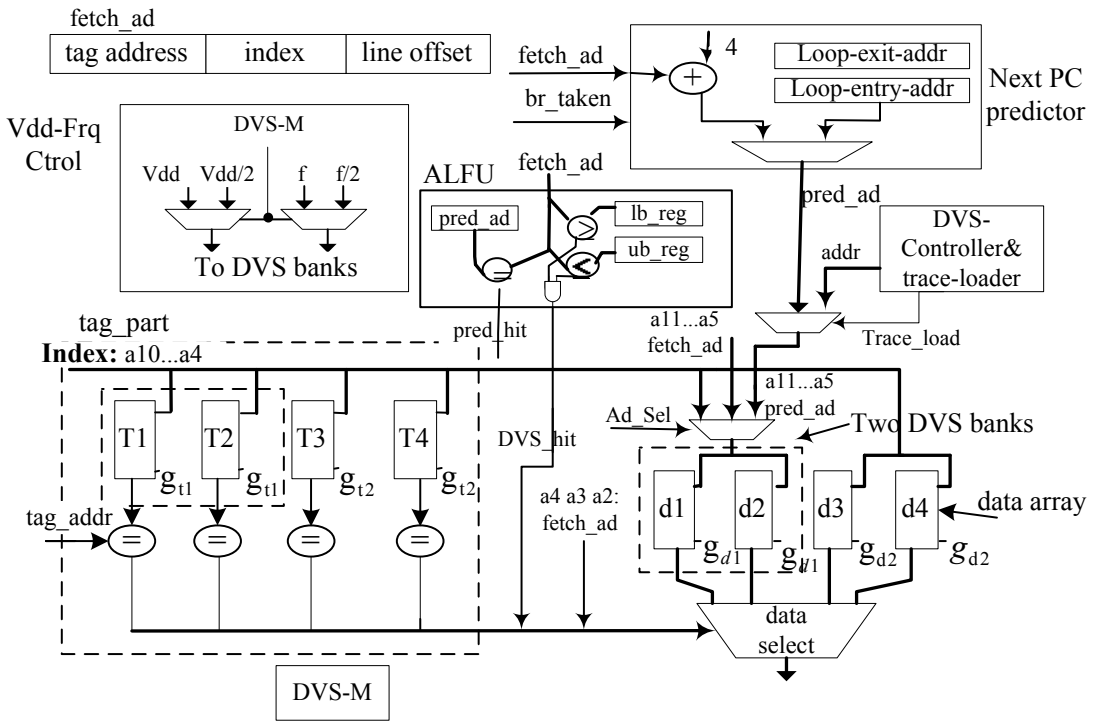

Figure 6.2: PRIM architecture.

a conventional cache, the supply voltage of banks d1 and d2 can be dynamically changed by the DVS controller. When a DVS reconfiguration instruction is encountered, the DVS controller first loads the appropriate instructions to the DVS banks. It then sets the DVS-M register, which in turn changes the the supply voltage and frequency of d1 and d2 to $\frac{V_{dd}}{2}$ and $\frac{f}{2}$, effectively reconfiguring the banks to DVS mode.

DVS controller is added to dynamically reload instructions to DVS banks when PRIM is configured as DVS mode during executions. DVS controller is essentially a component of Direct Memory Access (DMA) which is controlled by processor. It has two inputs, initial address of an instruction block and number of instructions to be loaded.

Instruction searching: The *address lookup functional unit* (ALFU) determines if an instruction is residing in the DVS banks or not. When two banks are configured to DVS mode, the tags corresponding to these two banks are com-

pletely disabled. In other words, the DVS banks act as software controlled memory. The address matching is accomplished inside the ALFU instead of tag matching in conventional caches. The ALFU contains two address registers and two parallel comparators. The two registers, `ub_reg` and `lb_reg`, hold respectively the upper and lower bound addresses for the instruction block that currently resides in the DVS banks. The value in `ub_reg` and `lb_reg` are updated when the DVS controller loads a block of instructions into the DVS banks. In the PRIM design presented here, there is only one pair of `ub_reg` and `lb_reg`. As a result, only one block of instructions may reside in the DVS banks at any point of time. If the address of the instruction to be fetched falls within the range of these two registers, then it resides in the DVS banks and the ALFU generates the `DVS_hit` signal. This signal controls the selection and gating of the tag and the data banks. To save dynamic energy, the two normal cache banks and their tags are not searched if the `DVS_hit` signal is asserted. On the other hand, the DVS banks are not searched if `DVS_hit` signal is not asserted. This is accomplished by the clock gating of the tag and data banks. The gating signals g_{t1}, g_{d1} and g_{t2}, g_{d2} are used to gate the clock of DVS banks and non-DVS banks, respectively:

$$g_{t1} = \text{DVS\_mode} \qquad g_{t2} = \text{DVS\_mode} \wedge \text{DVS\_hit}$$
$$g_{d1} = \text{DVS\_mode} \wedge \sim \text{DVS\_hit} \qquad g_{d2} = \text{DVS\_mode} \wedge \text{DVS\_hit}$$

As a result, in DVS mode, only two banks – either DVS banks or normal cache banks – are accessed at any point of time.

Pipelined parallel instruction fetches: In order to maintain the throughput of instruction fetches in DVS mode, when an instruction is fetched from one DVS bank, a speculative fetch is also started in the other DVS bank. The ALFU will later check to see if the speculation was successful. If the speculated PC equals that requested by the processor, the address prediction hit (pred_hit) is asserted to

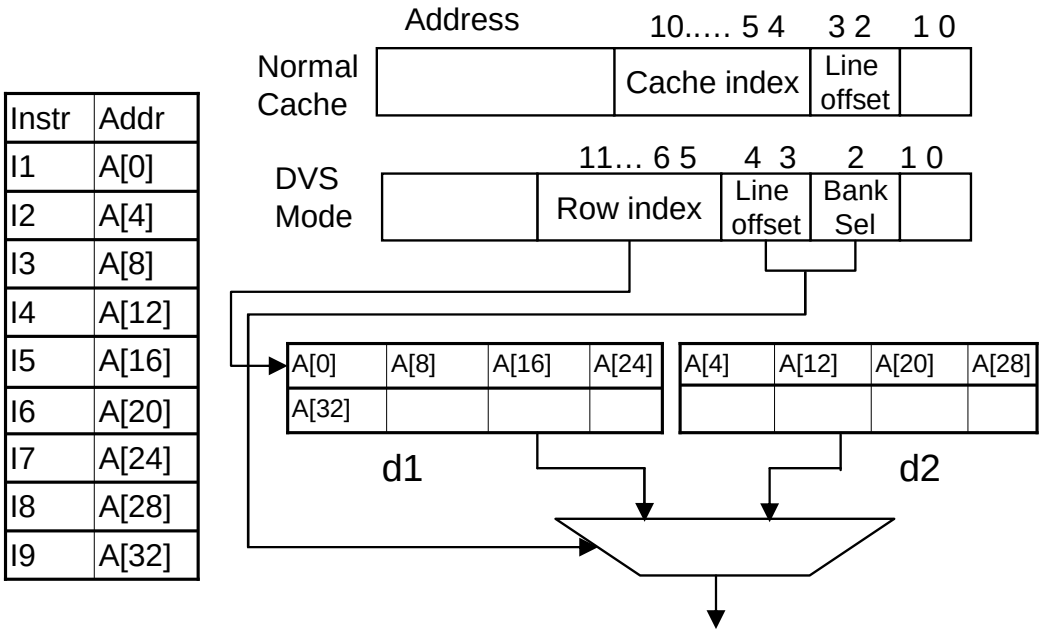

Figure 6.3: The layout of instructions

indicate a successful speculation. Otherwise, it is necessary to redo the instruction fetch, thereby incurring an overhead of twice the normal cache's latency.

From the above description, one can easily see why it is crucial to carefully layout the instructions in DVS banks so that sequentially executed instructions can be fetched in parallel from the two DVS banks. Figure 6.3 shows an example of the instruction layout. The left side of the figure is the instruction layout of a loop contain six instructions in main memory. Suppose that the sequence of instruction executed is $1-2-3-4-5-6-1\cdots$. In order to be able to fetch consecutive instruction in parallel, the odd and even numbered instructions need to be in different banks. We propose to layout the instructions in the way shown in right hand side of Figure 6.3. Bits 11...5 in Figure 6.3 are used to address the row of the DVS banks, while the column selection is determined by the least significant bit (bit 2 in Figure 6.3) of cache block index. The instruction selection from a cache line is decided by two bits 3 and 4.

Since the two DVS banks can be configured as two different types of instruction

buffer, namely conventional cache and software controlled memory, the address of two DVS banks need to be selectable from different address sources according to their mode. This is achieved by selection signal Ad_Sel. When applications try to load instruction block into DVS mode, Ad_Sel will select the address generated by 'DVS-Controller&trace-loader' component. If they are configured as DVS mode and operate on instruction fetching state, address a11...a5 will be chosen. Otherwise, the address (i.e cache index) will be selected for conventional cache mode.

The 'Next PC predictor' component of PRIM predicts the address of the next instruction to be fetched. If a branch is taken, we assume that we have encountered a loop and the address boundary will be dynamically written to registers 'Loop-entry-addr' and 'Loop-exit-addr'. If the current fetch address is not equal to the value in 'Loop-exit-addr', the next fetch address will be speculated to be the increment of the current fetch address. Otherwise, the next fetch address will be the value in 'Loop-entry-addr'.

6.3 Compilation framework

PRIM requires compiler support to achieve dynamic reconfiguration. The compiler decides which program regions are suitable for running in DVS mode, inserts appropriate instructions into the application to switch the two designated banks to DVS mode, and dynamically loads the selected instructions into the two DVS banks.

6.3.1 Compilation flow

The structure of our compilation flow is shown in Figure 6.4. The input are the application and the cache configuration. The output are the partitioning decisions

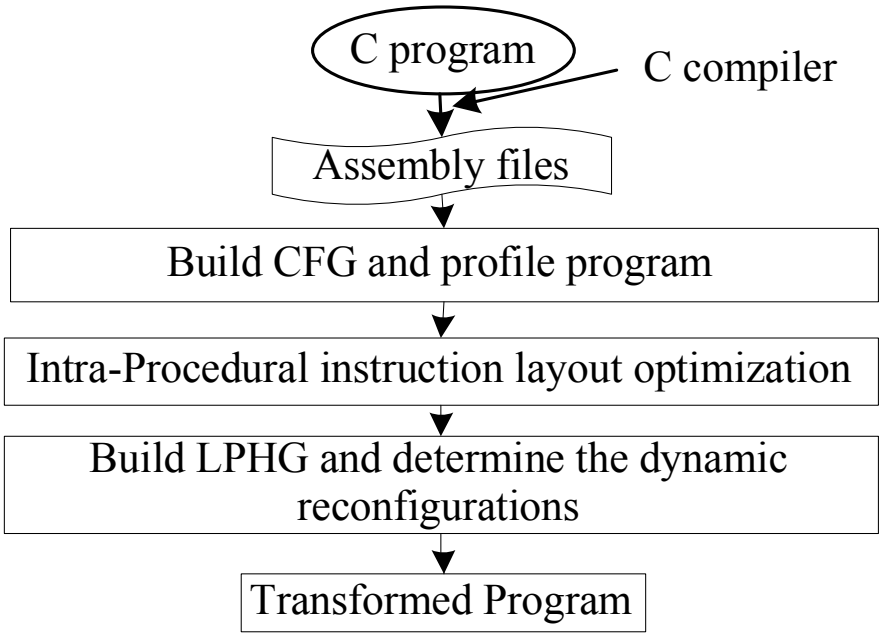

Figure 6.4: Design flow

for the instruction memory hierarchy for the application, and the transformed application with an optimized instruction layout. The framework consists of several steps:

- *Profiling the application*: First the application is profiled to collect the execution counts of the edges in the control flow graphs (CFG) of all the procedures and the number of invocations of each procedure.

- *Intra-procedural instruction layout optimization*: This step optimizes the instruction layout within each procedure to bring the frequently executed basic blocks together to make it easier to extract frequently executed traces.

- *Building the LPHG, and determining the reconfiguration and trace load points*: In this step, we build the Loop-Procedure Hierarchy Graph (LPHG) [64] to represent a program. The LPHG captures all the loops and procedure calls of an application as well as their relations. Loops are detected by building the dominator tree of the CFG. After the LPHG is built, the application

profile is analyzed to determine the suitable points to reconfigure PRIM. At the same time, the instruction traces to be loaded into the DVS banks at each reconfiguration point are also selected.

- *Inserting reconfiguration and trace load instructions*: Finally, the instructions for reconfiguration and trace loading are inserted into the application's code.

We have developed a prototype of this framework using the Simplescalar tool set [19]. The `sim-outorder` simulator is extended to support PRIM. The framework includes multiple tools to support profiling, intra-procedural instruction layout optimization, and identification of the reconfigurations and trace loading points.

6.3.2 Dynamic reconfigurations and instruction selection

This section describes the algorithm for deciding where and when to reconfigure PRIM as well as determining which instructions should go into the DVS banks. Our proposed algorithm is shown in Algorithm 6. We assume that most of the energy consumption due to instruction fetch occur inside the loops. The intuition is that if the time spent in a loop is long enough to hide the overhead of reconfiguration and instructions loading, then that loop is a candidate for allocation into the DVS banks. Thus we look for loops where the number of iterations exceeds a threshold. In this work, we empirically set the threshold value to be 20 iterations. If the loop size is too big to fit into the DVS banks, then the cache is used to buffer the rest of for the loop.

In a LPHG, the deeper a loop is, the higher is its execution frequency. The algorithm therefore starts from the leaf loops and works toward the root node. After a loop has been examined, its parent loop is added to *list_loops* (line 12). We may at some later point examine the parent node for further opportunities. If a loop

Algorithm 6: Algorithm for determining dynamic reconfiguration and DVS instruction load points

```
   Input: Proc_list: Procedure list whose procedures have been intra-
          procedural optimized
   Output: Basic_block_list: list of instruction blocks of basic blocks assigned
           to DVS banks
   BOOL conflict : if severe cache conflict incurred;
 1 Build Loop-Procedure Hierarchy Graph(LPHG)
 2 Get_sizes_of_all_loops();
 3 list_loops ⟵ all leaf loops;
 4 foreach loop l in list_loops do
       if l is leaf loop ⋀ #iterations of l ≥ Thresh_hold then
           Annotate_reconfig_point_and_instrs_partitioned(l);
 5     else if l is non-leaf loop then
 6         list_child_loops ⟵ all child loops of l;
 7         Update_iteration_num(list_child_loops);
 8         conflict=evalConflict(DVSmode, child_loops_of_l ∪ l);
 9         if !conflict then
10             Instr_alloc(list_child_loops ∪ l, DVS-banks);
           end
11     if !list_loops.contain(l) then
12         list_loops.push_back(l);
       end
   end
13 Hoist_reconfig_position();
14 Insert_reconfig_and_code_loading_instructions();
15 return Proc_list;
```

is an internal node (line 5), then the algorithm evaluates whether it is beneficial to configure the cache to DVS mode for its child loops (line 8). The evaluation function we use is conservative and simple. If the allocation of a child loop L to DVS banks does not adversely affect the miss rate of the other children, then the allocation is considered beneficial. In that case, the cache is configured to DVS mode at the entry point of the parent loop.

Figure 6.5 is an example of how the algorithm resolves conflicts, determines reconfiguration points, and selects the instructions for DVS banks. The left part of Figure 6.5 is a sample program represented in LPHG, while the right part is its corresponding CFG. The LPHG consists of the procedure P and five loops A, B, C,

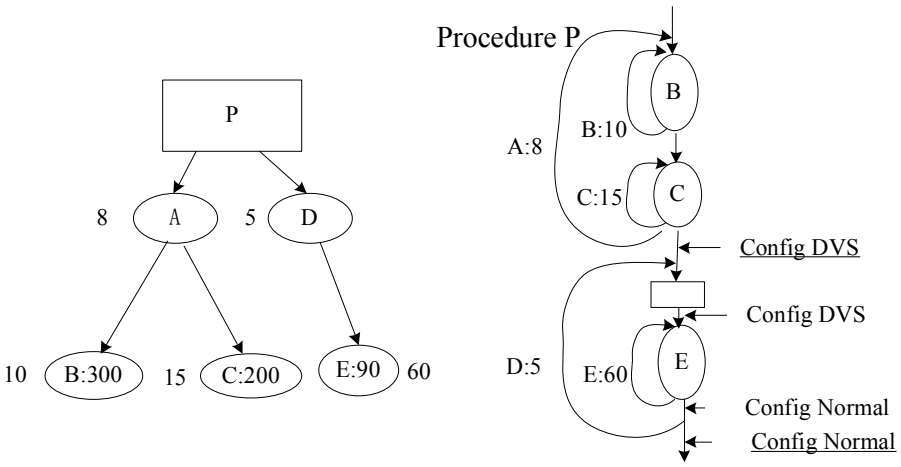

Figure 6.5: Code transformation for reconfiguration.

D, and E. The numbers beside the loops are the number of loop iterations while the numbers inside the loops are their sizes in terms of the number of instructions. Each of the four cache banks can hold up to 64 instructions. The algorithm first selects the three leaf loops. Only the number of iteration of E is larger than the threshold value (20). So the cache is configured to DVS mode at the entry point of E. We then traverse upward to the parent loops, A and D. As the number of iterations of D is smaller than the threshold, we will not use DVS mode for it. Loop A has two children, B and C, and the total number of iterations of B and C from the point of the entry of A is now larger than 20. The algorithm then check whether there will be severe cache conflicts if B or C is placed in the DVS banks. As the sizes of B and C are both larger than the remaining two normal cache banks, allocating either loop to the DVS banks will cause severe cache conflict to the other loop. Therefore, it is better to operate A in non-DVS mode.

The instruction allocation function assigns the frequently executed instructions inside the selected loop to the DVS banks (line 10). If the loop size exceeds the capacity of the DVS banks, then the most frequently executed instructions of the loop are allocated to the DVS banks.

Once all the loops have been processed and the reconfiguration points as well as instructions selection for the DVS banks have been finalized, the reconfiguration instructions have to be inserted. This is the same as that in compilation flow for DRIM design described in the previous chapter. There are two steps involved: optimization of hoisting the reconfiguration points and insertion of the reconfiguration instructions. The number of reconfigurations can be reduced by hoisting the reconfiguration point from the inner loop to the outer loop (line 13). If L is the only loop selected among its siblings, the reconfiguration point can be hoisted out from L to its parent loop. An example is shown in Figure 5.5 in the previous chapter. The DVS configuration instructions are hoisted to where they are underlined.

The last step in the algorithm (line 14) groups all the instructions allocated to the DVS banks at each reconfiguration point and inserts the instructions to reconfigure PRIM as well as load the instruction blocks into the DVS banks after each reconfiguration.

6.4 Experimental evaluation

6.4.1 Experimental methodology

We used the Simplescalar-PISA 3.0d simulator [19] for our experiments. The full-featured simulator `sim-outorder` was modified to support PRIM. The cache line used in the simulation corresponds to 32 bytes in a 4-byte word processor. The instruction memory hierarchy consists of the L1 instruction buffer (i.e. PRIM) and the main memory. Our PRIM implementation has four banks of data storage, each capable of holding 64 instructions. The latency of accessing L1 cache is 1 cycle. The main memory is assumed to be pipelined. The latency of the first access to the

main memory is 10 cycles, while that of the subsequent accesses is 2 cycles. In our simulation, the Trace_load instructions used to load instruction blocks to DVS banks are compiled to binary code. As a result, the overhead of execution time and cache miss rate increase caused by these instructions are included. The time for loading a trace into the DVS banks is calculated as the time to perform the necessary burst accesses to the SDRAM to bring in the entire trace.

In our experiments, we used six application benchmarks from the MediaBench [61] and MiBench [32] suites. We compared the energy consumption and performance across three different configurations: (1) a baseline system comprising of a traditional 4-way associative instruction cache, (2) a simple low V_{DD} scheme (Low-VDD) (3) a selective way cache [6] (Sel-Way), and (4) PRIM. We use two voltage settings, 0.5×Vdd and 0.8×Vdd, for different process technology of CPUs. The frequency will be always halved for both voltage options. We chose selective way cache for comparison because it disables and puts under-utilized memory banks into low power mode to save energy. In our experiments, Sel-Way gates the clock of a certain number of cache banks if it is under-utilized. To do this, we analyze the LPHG and put frequently executed loops into certain cache banks and gate the rest of the cache banks. The instructions of the frequently executed loops are locked for a certain time interval to avoid unintentional replacement. Low-VDD scheme just simply halves supply voltage and clock frequency of L1 instruction cache to save energy while the access latency to the instruction cache is doubled.

6.4.2 Energy calculations

In this work, our focus is on reducing dynamic energy consumption. For power constrained embedded processors, we believe that dynamic energy consumption will

continue to dominate static energy consumption. We modeled the energy consumption of the memory hierarchy using the CACTI [99] model for $0.13\mu m$ technology.

To calculate the energy consumption of PRIM, we included the extra logic elements not found in a normal cache, namely the ALFU, the Next PC predictor and the two extra multiplexers. We assume that the energy consumed by the PRIM's comparators and multiplexers are the same as the multiplexers and comparators components of the normal cache assumed by CACTI. We approximated the energy consumption of an adder or subtractor in ALFU as that of a multiplexer. These hardware components are always powered at regular voltage.

According to CACTI, the total per-access energy of a n way cache is as follows:

$$e_C = e_t + e_d + e_{com} + e_{mux} + e_{out}$$

where e_t, e_d, and e_{com} stand for the energy per access to all tags, data storages, and comparators, respectively. e_{mux} is the energy consumption of the multiplexer, while e_{out} is the energy of the output driver. We rewrite this formula as:

$$e_C = n \times e_b + e_{mux} + e_{out}$$

where e_b is the energy per access for one cache bank. This includes the per-access energy for the data and tag of one bank, and a comparator.

According to general energy equation, i.e., $E = C \times V^2 \times f \times t$, when supply voltage and frequency is decreased to V', the energy consumption becomes $(\frac{V'}{V})^2 \times 2f$ (we define this factor as C_f) of the original value. We assume that the energy of the

data output drivers of PRIM will be the same as that of a normal cache since this component has to be powered at the regular voltage while that for the cache banks, comparators and multiplexer is decreased. There are three ways in which PRIM is accessed:

1. When PRIM is in DVS mode and the instruction is in one of the DVS banks, PRIM can deliver it directly from the DVS bank without searching the other DVS bank. Thus, energy of an instruction fetch is:

$$e_D = \frac{e_b + e_{mux}}{C_f} + e_{out} + e_O$$

   where e_O is the energy overhead per access due to the extra hardware components described above. It is about 3.5% of the total per-access energy of the baseline cache.

2. When PRIM is in DVS mode and the instruction is not in a DVS bank, the other two cache banks powered at the regular Vdd, will be searched. The per-access energy for this is:

$$e_{\neg D} = 2 \times e_b + e_{mux} + e_{out} + e_O$$

3. If PRIM is in normal cache mode, then energy consumption is simply the energy of the baseline cache plus the energy overhead:

$$e_N = e_C + e_O$$

The last two energy components of PRIM are the energy consumed during the loading of traces, E_L, into the DVS banks, and the energy caused by cache misses. E_L is approximated by:

$$E_L = \sum_{l=0}^{L} \frac{s_l}{B} \times e_M$$

where L is the total number of traces loaded, s_l is the size of trace l and B is the size of a cache block. e_M is the energy penalty of a cache miss. Its value is assumed 50 times of energy consumption of one access to four way baseline cache [103]. Since the DVS controller is only active during instruction trace loading to PRIM, and because the energy of a cache miss is orders of magnitude greater, we did not include it in the energy calculations. The total overall energy consumption due to memory accesses to PRIM is given by:

$$E_{\text{PRIM}} = A_D \times e_D + A_{\neg D} \times e_{\neg D} + A_N \times e_N + A_M \times e_m + E_L$$

where A_D and $A_{\neg D}$ are the numbers of accesses to the DVS banks and two cache banks respectively when PRIM is in DVS mode, while A_N is the number of accesses to PRIM when it is in normal cache mode. A_M is the number of cache misses.

As with PRIM, for Low-Vdd cache, the voltage of data output driver is assumed normal and the energy consumption of this component remains same as that of baseline cache. For Sel-WAY cache, the energy is depend on the number of active cache banks, n, which is shown as follows:

$$e_S = n \times e_b + e_{mux} + e_{out}$$

Table 6.1 shows the energy consumption per access to different architectures. The result of our energy calculation is consistent with the energy consumption of 32K L1 cache in [59] in which SPICE was used to simulate the energy consumption.

According to [59], the total energy consumption of 32K L1 cache powered at 1V is 1.055nJ while the energy at 0.5V is 0.1250nJ.

		PRIM				Sel-Way			
base cache (e_C)	Low-Vdd	$e_{\neg D}$	e_N	e_D (0.5×Vdd)	e_D (0.8×Vdd)	1way	2way	3way	4way
0.538	0.144	0.296	0.557	0.065	0.165	0.146	0.277	0.407	0.538

Table 6.1: Per-access energy consumption (in nJ).

6.4.3 Experiment results

Performance: The performance results are shown in Table 6.2. 'fetch miss' represents the miss rate of instruction fetch while 'perform overhd' stands for the performance overhead compared to the baseline. 'Pred-miss' is the miss rate of address prediction of the next speculatively fetched instruction in DVS mode. Compared to the baseline cache configuration, we can see some miss rate reduction achieved by PRIM and Sel-Way schemes for certain benchmarks. We attribute this to the locking of frequently executed instructions, thereby reducing conflict misses. The miss rate of address prediction ranges from 0.63% to 8.45% with an average value of 2.85%. The 'pred-miss' rate for mpeg2-dec and pegwit are quite high. The loops in mpeg2-dec have many procedure calls resulting in poor locality. For pegwit, there are many control flow transfers inside the loops which is detrimental to the prediction process.

Benchmark	Baseline	Low-Vdd	PRIM			Sel-Way	
	fetch miss(%)	perform overhd(%)	fetch miss(%)	pred-miss (%)	perform overhd(%)	fetch miss(%)	perform overhd(%)
mpeg2-dec	1.35	107.0	1.30	4.8	-0.03	1.49	3.07
gsm-dec	0.42	121.8	0.40	0.71	0.20	0.41	0.07
susan-edge	2.76	80.9	2.36	1.11	-6.65	2.15	-13.5
FFT	0.06	146.5	0.09	1.42	2.82	0.09	0.85
pegwit	0.38	45.8	0.35	8.45	0.78	0.38	3.54
sha	0.054	252.1	0.35	0.63	1.66	0.046	3.85
Average	-	125.6	-	2.85	-	-	-

Table 6.2: Miss rate and performance overhead

The performance overhead varies across different applications. There are mainly

four factors affecting the overhead: the latency of cache accesses, the cache miss rate, the next address prediction miss rate, and the overhead of cache reconfigurations. As we can see from Table 6.2, Low-Vdd suffers from a high performance overhead that averages 125.6% because of the doubling of the latency in the instruction cache. PRIM, on the other hand, has a low performance overhead ranging from −6.65% to 2.82%. For the benchmark susan-edge, performance is in fact improved because of the reduction in the cache miss rate. On the other hand, for the benchmark pegwit, the poorer performance is due to the penalty of pred-miss and reconfiguration exceeding the gains from cache miss rate reduction.

Energy consumption: The total dynamic energy consumption of the four instruction memory hierarchies are shown in Table 6.3. We evaluated two PRIM configurations – one that halves Vdd ('0.5×Vdd ') and one that lowers Vdd by 20% ('0.8×Vdd '). In both these configurations, the operating frequency is halved, doubling access times to these banks. All the schemes achieve significant energy savings compared to the baseline. The reduction in the average energy consumption achieved by the Sel-Way cache is 39.0% for the benchmarks studied, while that for PRIM is 56.6% at 0.5×Vdd and 45.1% at 0.8×Vdd. In other words, PRIM at 0.5×Vdd and 0.8×Vdd achieved 17.6% and 6.1% more energy saving than Sel-Way respectively. Energy savings of PRIM at 0.8×Vdd is smaller than PRIM at 0.5×Vdd because the supply voltage scaling down is more conservative.

Low-Vdd achieved an average 56.5% energy savings which is almost the same as PRIM at 0.5×Vdd. However, Low-Vdd comes with significant performance loss as shown in Table 6.2 because of the doubling of the cache hit latency. As the result, the additional energy consumed by other parts of a processor can surpass the energy savings obtained by instruction memory hierarchy.

Energy savings come mainly from DVS mode access, clock gating of underutilized ways, and cache miss rate reduction. Operating in the DVS mode, PRIM

Benchmark	Baseline	Low-Vdd(0.5×Vdd)			PRIM(0.5×Vdd)			PRIM(0.8×Vdd)			Sel-Way		
	En (mJ)	En (mJ)	impr (%)	En*delay	En (mJ)	impr (%)	En*delay	En (mJ)	impr (%)	En*delay	En (mJ)	impr (%)	En*delay
mpeg2-dec	34.5	19.43	43.7	1.17	25.6	25.8	0.74	27.55	20.1	0.80	30.26	12.3	0.90
gsm-dec	8.10	3.20	60.5	0.88	3.66	54.7	0.45	4.61	43.0	0.57	4.35	46.3	0.54
susan-edge	2.92	2.02	30.8	1.25	1.89	35.2	0.60	2.00	31.4	0.64	1.98	32.3	0.59
FFT	1.06	0.31	71.0	0.71	0.21	80.0	0.21	0.40	62.0	0.39	0.51	51.5	0.49
pegwit	19.78	7.63	61.5	0.56	7.78	60.7	0.40	10.19	48.5	0.52	11.21	43.4	0.59
sha	7.53	2.16	71.3	1.01	1.25	83.4	0.17	2.58	65.8	0.35	3.89	48.4	0.54
Average	-	-	56.5	0.93	-	56.6	0.43	-	45.1	0.54	-	39.0	0.61

Table 6.3: Energy consumption.

can save more energy than the clock gating used in Sel-Way. This is evident in the benchmark FFT where the improvement of PRIM over Sel-Way scheme is very significant. In the case of FFT, the cache miss rate is very low and so cache misses contribute only a small amount of total energy consumption. The main contributor to the total energy is the on-chip cache access energy which is significantly reduced in PRIM. For the benchmark susan-edge, the energy improvement of PRIM over Sel-Way is small. This benchmark has a big loop with a large number of iterations that is larger than the cache. As a result, it is beneficial to allocate as many cache entries as possible and lock the instructions of this loop in these cache entries to avoid cache conflict. Sel-Way allocates and locks three cache banks for the instructions in the loop, while our PRIM configuration can only allocate two cache banks in DVS mode. Sel-Way is therefore able to avoid more cache conflicts in this benchmark.

Energy delay product: 'En*delay' in Table 6.3 represents normalized energy delay product relative to the baseline cache. As shown in the table, PRIM at 0.5×Vdd, PRIM at 0.8×Vdd, Low-Vdd and Sel-Way achieved average normalized energy delay product value 0.43, 0.54, 0.93, and 0.61 respectively. Although Low-Vdd obtained significant energy savings to memory hierarchy, the energy delay product is very close to that of the baseline cache because of the large performance overhead. PRIM at 0.5×Vdd and 0.8×Vdd achieved the best energy-delay product since it can effectively decrease energy consumption with only a slight performance loss.

6.5 Summary

Shutting down idle cache banks is one of the most popular methods to reduce the energy consumption. However, the capacity of the idle cache banks is wasted. In this chapter, we proposed PRIM, a pipelined, DVS-based dynamically reconfigurable instruction memory hierarchy to exploit the under-utilized storage resource to achieve more energy savings. Instead of shutting down unused storage bank, the unused storage bank and a normal bank in PRIM can be reconfigured as a low-V_{dd}, low frequency loop buffer pair to obtain more energy saving for instruction fetching. To maintain the throughput of instruction fetch, pipelined speculative reads to these two banks are used when accessing this buffer in the low-V_{dd} mode. Through this method, the capacity of the unused bank is exploited.

We also developed a compilation flow to support PRIM. Our experimental results showed significant energy savings while performance is affected only marginally. PRIM also performs better than selective way caches in overall energy saving. Although we have evaluated PRIM using a 4-bank cache structure, it is certainly possible to extend this to other numbers of banks. We believe that PRIM or PRIM-like techniques can easily be applied to embedded processors and is valuable as a contribution to the repertoire of techniques for energy-aware computing.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this thesis, we studied the instruction memory hierarchy of embedded systems with the aim of achieving performance improvement and energy savings. The instruction memory hierarchy is of crucial importance for embedded computing systems. Our study focused on the three schemes: SRIM, DRIM and PRIM. Our study showed that the proposed schemes achieved significant performance improvement and energy savings over traditional cache architectures. The motivation and contribution of our designs can be summarized as follows:

- **SRIM:** The SRIM architecture design can be used to utilize a limited resource budget for instruction memory hierarchy to achieve performance improvement and energy savings. We first showed that different applications require distinct architectural configurations instead of using purely a cache to obtain higher performance. The reason for this is that different applications have distinct characteristics, which in turn gives rise to the need of tuning architectural

parameters to suit the applications. Due to the flexibility in the hardware resource usage, our experimental results demonstrated that SRIM has significant performance improvement and energy savings over the traditional cache architecture. This performance improvement comes from mapping the frequently executed instructions into SPM, which may decrease the cache conflict misses. As a result of the reduction of the cache miss rate, the execution times were also improved. There are two main factors that contribute to the energy savings. The first one is the reduction of cache fetch misses. The accesses to off-chip main memory consume a large amount of energy and thus a small amount of cache miss reduction can achieve great energy savings. The other factor is that scratch-pad memory is more energy efficient than cache architectures. Therefore, we can obtain energy savings by mapping the most frequently executed instructions into SPM.

Apart from reconfiguring architectural parameters, application optimizations such as instruction layout optimization are also widely used. In this thesis, we also studied the interactions between these two methods and integrated architecture exploration and instruction layout optimization into one unified framework to achieve more performance and energy improvement. We achieved significant performance improvement and we also concluded that the sequence of performing these two methods could result in different performance and energy consumption.

- **DRIM:** The DRIM design explores dynamic architecture reconfigurations and instruction replacement, and it is more flexible than SRIM design. The advantage of SRIM method is that the architectural parameters can be tuned for specific applications according to their characteristics. However, there are two main drawbacks of this static method: 1) SPM is not efficiently used as it cannot be reused by multiple instructions. 2) Architectural parameters cannot

be changed at different phases of an execution for more energy savings. In order to overcome the above two limitations, we should develop a method which can dynamically replace the instructions allocated to SPM and reconfigure the architectural parameters. DRIM was design to handle this issue. We obtained an average of 15.6% improvement in the miss rate and an average of 10.2% performance improvement for six benchmarks selected. This improvement comes from reconfiguring some storage banks to SPM and mapping of the frequently executed instructions into the SPM for important loops. The average energy savings was 41%. The reasons for the energy consumption improvement are the reduction of instruction fetch misses and the better efficiency in energy consumption of SPM.

- **PRIM:** The PRIM design exploits the under-utilized resource to achieve energy savings. One of the most popular methods to reduce energy consumption is to shut down the under-utilized resources. The drawback of this method is that the capacity of under-utilized resources is wasted. To fully exploit the potential of under-used resources to achieve more energy savings, we proposed and designed PRIM architecture. The under-used storage bank and a normal storage bank can be reconfigured as a low-V_{dd} mode pair. The energy consumption per instruction fetch in low power mode will be less than that of one bank in normal mode. To maintain the throughput, instruction fetches to these two banks are pipelined. We obtained significant energy savings over traditional architectures, way-shutdown cache and normal cache, for six benchmarks selected while performance was affected only marginally.

In this thesis, we have demonstrated that customization of instruction memory hierarchies for embedded systems can achieve significant performance improvement and energy savings. Our methods are novel in methodology and advance the state of art. In addition, we have also implemented SRIM in real hardware which have rarely

been done by other researchers. We believe that the techniques we proposed in this thesis will contribute to more energy efficient and better performance embedded systems.

7.2 Future work

Our proposed schemes have achieved significant performance and energy improvement. Based on the current work, the following can be considered.

First of all, one of the limitations of our methods is that the designs are profile driven. Our methods need to profile applications to get the execution statistics for a certain input and then determine the architectural reconfiguration and instruction partitioning based on the obtained profiling information. The drawback is that the execution statistics may vary across different inputs and thus the reconfiguration and instruction partitioning determined based on one input may not be suitable for others. Some researchers have addressed this issue of the limitations of profiling based designs. However, it has not been solved yet and remains an open problem. Further study can be carried out to address the issue of how the execution statistics vary across different inputs and how to avoid the negative effects of this variation.

Another research direction is the reconfigurable instruction memory hierarchy for multi-core processor systems(Multi-cores). Multi-cores are becoming increasingly popular for high performance computing systems in recent years. Similar to single core processor systems, the memory subsystem of Multi-cores is the bottleneck for both performance and energy. Moreover, there are more challenging issues involved in designing memory subsystem for Multi-cores as Multi-cores are more complicated than single core process systems. Thus, intelligently managing memory subsystem is of crucial importance to achieve high performance and low energy consumption.

Further research can be carried out to extend our reconfigurable instruction memory schemes to Multi-cores for performance and energy improvement.

Bibliography

[1] Microblaze processor reference guide. In *www.xilinx.com*.

[2] Nios development board reference manual. In *www.altera.com*.

[3] Tms320vc5501 fixed-point digital signal processor data manual. In *http://focus.ti.com/lit/ds/symlink/tms320vc5501.pdf*.

[4] 68HC12BC32 Data Sheet. `http://www.freescale.com/files/microcontrollers/doc/data_sheet/M68HC12B.pdf?pspll=1`.

[5] T. V. Aa, M. Jayapala, F. Barat, G. Deconinck, R. Lauwereins, F. Catthoor, and H. Corporaal. Instruction buffering exploration for low energy vliws with instruction clusters. In *Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 824–829, 2004.

[6] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture(MICRO 32)*, pages 248–259, 1999.

[7] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioninge. In *Proc. of the 2003 Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '03)*.

[8] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proc. of the 2004 Intl. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '04)*, pages 259–267, September 2004.

[9] ARM. http://www.arm.com/products/cpus/arm1156t2-s.html.

[10] O. Avissar and R. Barua. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.

[11] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the conference on Design, automation and test in Europe*, 2002.

[12] R. Babakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratch-pad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78, 2002.

[13] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33th annual ACM/IEEE international symposium on Microarchitecture*, 2000.

[14] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. A dynamically tunable memory hierarchy. *IEEE Tansactions on Computers*, 52(10):1243–1258, May 2003.

[15] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. pages 449–453, 2002.

[16] L. Benini, A. Macii, E. Macii, and M. Poncino. Minimizing memory access energy in embedded systems by selective instruction compression. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 521 – 531, 2002.

[17] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A dynamic voltage scaled microprocessor system. In *Proceedings of IEEE International Solid-State Circuits Conference*, 2000.

[18] T. D. Burd and R. W. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the 2000 international symposium on Low power electronics and design*, pages 9 – 14, 2000.

[19] D. Burger and T. M. Austin. The Simplescalar tool set, version 2.0. *Technical Report #1342, University of Wisconsin-Madison Computer Sciences Department*, May 1997.

[20] B. Calder, D. Grunwall, , and J. Emer. redictive sequential associative cache. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, 1996.

[21] F. Catthoor, S. Wuytack, E.D. Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Kluwer Academic Publisher, 1998.

[22] S. Chandar, M. Mehendale, and R. Govindarajan. Area and power reduction of embedded dsp systems using instruction compression and re-configurable encoding. In *Journal of VLSI Signal Processing Systems*, pages 245 – 267, 2006.

[23] A. Chandraksan and R. Brodersen. Lower power digital CMOS design. In *Kluwer Academic Publishers*, 1995.

[24] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95 – 105, 2002.

[25] T. Van der Aa, M. Jayapala, F. Barat, G. Deconinck, R. Lauwereins, H. Corporaal, and F. Catthoor. Instruction buffering exploration for low energy embedded processors. In *Proceedings of International Workshop on Power And Timing Modeling Optimization and Simulation*, September 2003.

[26] Transmeta. Crusoe. Processor for Embedded Applications. http://datasheets.chipdb.org/transmeta/pdfs/brochures/embedded_apps_030904.pdf.

[27] Z. Ge, H.B. Lim, and W.F. Wong. A reconfigurable instruction memory hierarchy for embedded systems. In *Proceedings of the 15th International Conference on Field Programmable Logic and Applications*, pages 7–12, 2005.

[28] Z. Ge, W.F. Wong, and H.B. Lim. DRIM: A low power dymaically reconfigurable instruction memory hierarchy for embedded systems. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, 2007.

[29] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *Proc. of Micro-30*, pages 303–313, December 1997.

[30] A. Gordon-Ross, S. Cotterell, and F. Vahid. Tiny instruction caches for low power embedded system. *ACM Transactions on Embedded Computing Systems*, 2(4):449–481, November 2003.

[31] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, 1995.

[32] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[33] Y. Han. In *http://www.ece.neu.edu/groups/nucar/barc2004/BARC2004-yhan.pdf*, 2004.

[34] A. Hasegawa, I.Kawasaki, K.Yamada, S.Yoshioka, S. Kawasaki, and P. Biswas. Sh3: High code density, low power. In *Proceeding of Annual International Symposium on Microarchitecture*, 1995.

[35] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *Proc. of PLDI'97*, pages 171–182, 1997.

[36] J. L. Hennessy and D. A. Patterson. Computer architecture: A quantitative approach. In *Morgan Kaufmann*, 2003.

[37] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 35th annual conference on Design automation*, pages 176 – 181, 1998.

[38] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scheduling. In *Workshop on Power-Aware Computer Systems (PACS)*, 2000.

[39] C.-H. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[40] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units.

In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2004.

[41] C. Huang, S. Ravi, A. Raghunathan, and N.K. Jha. High-level synthesis using computation-unit integrated memories. In *Proceedings of the International Conference on Computer-Aided Design*, pages 783–790, 2004.

[42] M. Huang, J. Renau, S.M. Yoo, and J. Torrellas. L1 data cache decomposition for energy efficiency. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[43] K. Inoue, T. Ishihara, and K. Murakami. Way-predictive set-associative cache for high performance and low energy consumption. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 1999.

[44] K. Inoue, K. Kai, and K. Murakami. Dynamically variable line-size cache exploiting high on-chip memory bandwidth of merged dram/logic lsis. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture(HPCA)*, 1999.

[45] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, page 197C202, 1998.

[46] P. Jain, G. Edward Suh, and S. Devadas. Embedded intelligent sram. In *Proceedings of the 40th conference on Design automation(DAC)*, pages 869–874, 2003.

[47] A. Janapsatya, S. Parameswaran, and A. Ignjatovic. Hardware/software managed scratchpad memory for embedded system. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, 2004.

[48] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *25 Years ISCA: Retrospectives and Reprints*, pages 388–397, 1998.

[49] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 628–633, 2002.

[50] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture(MICRO-31)*.

[51] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the International Symposium on Computer Architecture(ISCA)*, 2001.

[52] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. In *ACM SIGARCH Computer Architecture News*, 1989.

[53] H. Kim, A.K. Somani, and A. Tyagi. A reconfigurable multifunction computing cache architecture. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2001.

[54] K. Flautner; N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the 29th International Symposium on Computer Architecture(ISCA 29)*, 2002.

[55] N. S. Kim, D. Blaauw, and T. Mudge. Quantitative analysis and optimization techniques for on-chip cache leakage power. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1147 – 1156, 2005.

[56] N. S. Kim, K. Flautner, D. Blaauw, and T. Mudge.

[57] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Mooreapos;s law meets static power. In *IEEE Trans. Computer*, pages 68 – 75, 2003.

[58] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[59] C.K. Koh, W.F. Wong, Y. Chen, and H. Li. Vosch: Voltage scaled cache hierarchies. In *Proceedings of International Conference on Computer Design*, 2007.

[60] M. Kondo, H. OKAWARA, H. NAKAMURA, and T. BOKU. SCIMA: Software controlled integrated memory architecture for high performance computing. In *Proceedings of the 2000 International Conference on Computer Design*, pages 105–111, 2000.

[61] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: a tool for evaluating multimedia and communications systems. *Proceedings of the 30th IEEE/ACM International Symposium on Microarchitecture(MICRO)*, 1997.

[62] L. H. Lee, W. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings of International Symposium on Low Power Electronics and Design*, 1999.

[63] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th conference on Design automation(DAC)*, 2000.

[64] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In

DAC '00: Proceedings of the 37th conference on Design automation, pages 507–512, 2000.

[65] embedded processor lpc3180. http://www.standardics.nxp.com/products/lpc3000/datasheet/lpc3180.pdf.

[66] P. Machanick. Approaches to Addressing the Memory Wall. Technical report, School of IT and Electrical Engineering, University of Queensland, 2002.

[67] S. McFarling. Program optimization for instruction caches. In *ACM SIGARCH Computer Architecture News*, pages 183 – 191, 1989.

[68] Intel XScale microarchitecture. http://developer.intel.com/design/intelxscale/.

[69] D. Nicolaescu, X. Ji, A. V. Veidenbaum, A. Nicolau, and R. K. Gupta. Compiler-directed cache line size adaptivity. In *Intelligent Memory Systems*, 2000.

[70] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems.*, 6(2):149–206, 2001.

[71] P. R. Panda, N. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. In *ACM Transactions on Design Automation of Electronic Systems (TODAES).*

[72] P. R. Panda, N. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. *ACM Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 5(3), January 1999.

[73] P. R. Panda, N. Dutt, and A. Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):682–704, July 2000.

[74] P.R. Panda, N. Dutt, and A. Nicolau. *Memory Issue in Embedded Systems-on-chip: Optimization and Exploration*. Kluwer Academic Publisher, 1999.

[75] K. Pettis and R. C. Hansen. Profiling guided code positioning. In *Proc. of PLDI'90*, pages 16–27, 1990.

[76] C. Piguet. Low-power processors and systems on chips. In *CRC Press, Inc. Boca Raton, FL, USA*, 2005.

[77] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and N. Vijaykumar. Reducing leakage in a high-performance deep-submicron instruction cache. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 77 – 89, 2001.

[78] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T.N. Vijaykumar. Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2000.

[79] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy viaway-prediction and selective direct-mapping. In *Proceeding of 34th Annual International Symposium on Microarchitecture*, 2001.

[80] P. Ranganathan, S. Adve, and N. P. Jouppi. Reconfigurable caches and their application to media processing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 214–224, 2000.

[81] R. A. Ravindran. Compiler managed dynamic instruction placement in a low-power code cache. In *Proceedings of the international symposium on Code generation and optimization*, pages 179–190, 2005.

[82] R. A. Ravindran, P. D. Nagarkar, G. S. Dasika, E. D. Marsman, R. M. Senger, S. A. Mahlke, and R. B. Brown. Compiler managed dynamic instruction placement in a low-power code cache. In *Proceedings of the international symposium on code generation and optimization*, pages 179–190, 2005.

[83] S. Segars, K. Clarke, and L. Goudge. Embedded control problems, thumb, and the arm7tdmi. In *Micro, IEEE*, pages 22 – 30, 1995.

[84] S.-W. Seong and P. Mishra. An efficient code compression technique using application-aware bitmask and dictionary selection methods. In *Proceedings of the conference on Design, automation and test in Europe*, pages 582 – 587, 2007.

[85] G. Sery, S. Borkar, and V. De. Life is cmos: why chase the life after? In *Proceedings of the 39th conference on Design automation*, pages 78 – 83, 2002.

[86] T. Sherwood and B. Calder. Time varying behavior of programs. In *UC San Diego Technical Report UCSD-CS99-630*, 1999.

[87] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 336–349, 2003.

[88] D. Shin and J. Kim. A profile-based energy-efficient intratask voltage scheduling algorithm for hard real-time applications. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[89] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 134 – 139, 1999.

[90] A. Shrivastava, I. Issenin, and N. Dutt. Compilation techniques for energy reduction in horizontally partitioned cache architectures. In *Proceedings of*

the 2005 international conference on Compilers, architectures and synthesis for embedded systems, pages 90–96, 2005.

[91] J. Sjodin, B. Froderberg, and T. Lindgren. Allocation of global data objects in on-chip ram. In *Proceeding of Workshop on Compiler and Architecture Support for Embedded Computing Systems*, 1998.

[92] J. Sjodin and C. V. Platen. Storage allocation for embedded processors. In *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, 2002.

[93] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, pages 213–218, Kyoto, Japan, October 2002.

[94] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scatchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, pages 409–416, 2002.

[95] A. V. Veidenbaum, W. Tang, and R. Gupta. Adapting cache line size to application behavior. In *Proceedings of the 13th International Conference on Supercomputing*, pages 145–154, 1999.

[96] M. Verma and P. Marwedel. Advanced memory optimization techniques for low-power embedded processors. In *Springer*, 2007.

[97] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21264, Washington, DC, USA, 2004. IEEE Computer Society.

[98] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104 – 109, October 2004.

[99] S. J. E. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[100] A. Wolfe and A. Chanin. Executing compressed programs on an embedded risc architecture. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 81 – 91, 1992.

[101] M. Wolfe. High performance compilers for parallel computing. In *Addison-Wesley Publishing Company*, 1996.

[102] Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa. An object code compression approach to embedded processors. In *Proceedings of the 1997 international symposium on Low power electronics and design*, pages 265 – 268, 1997.

[103] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 136–146, 2003.

[104] W. Zhang, J.S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin. Compiler-directed instruction cache leakage optimization. In *Proceedings of the International Symposium on Computer Architecture(ISCA 29)*, 2002.